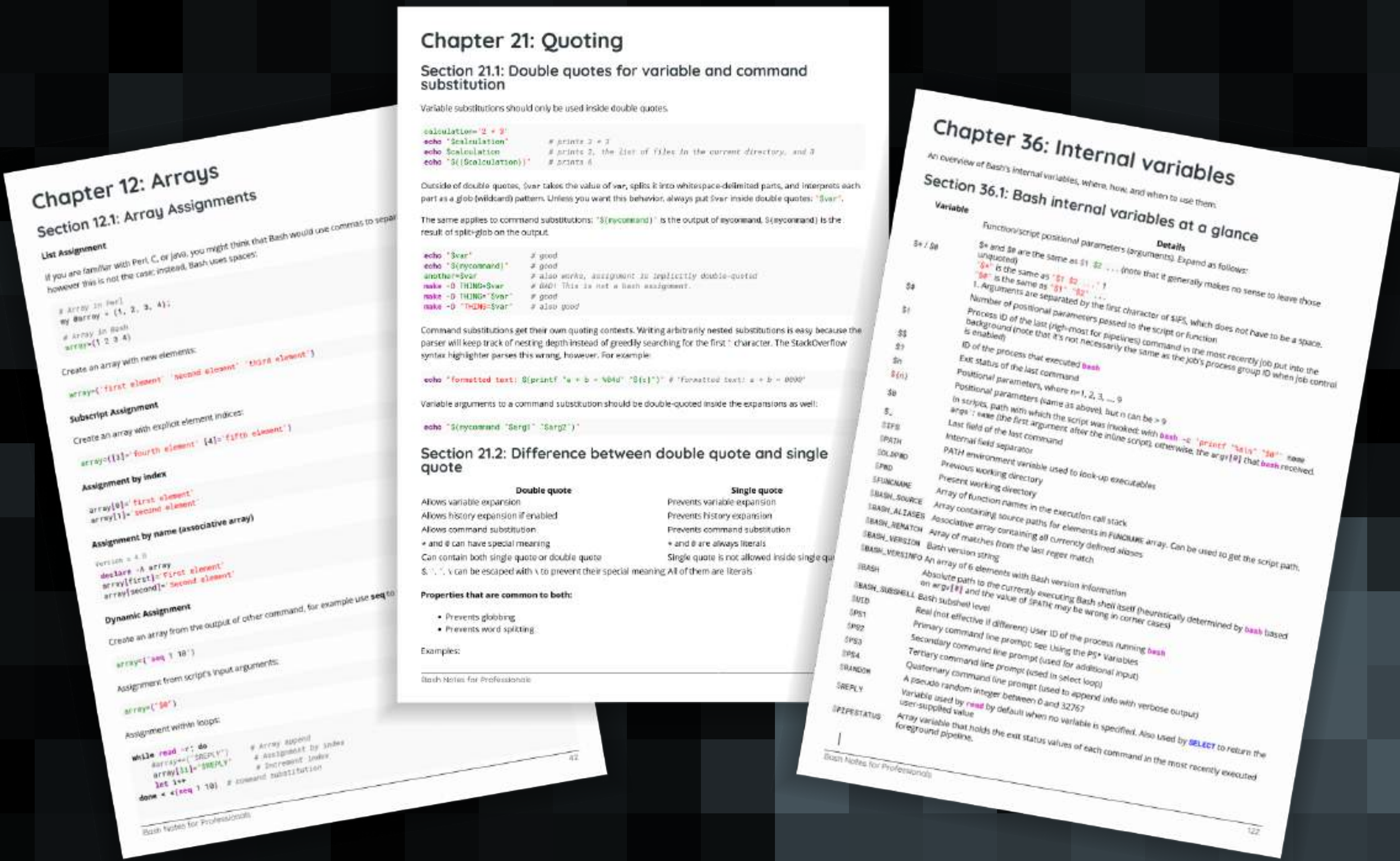
Bash

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Contents

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/BashBook

This *Bash Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow.

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

Chapter 1: Getting started with Bash

Version	Release Date
0.99	1989-06-08
1.01	1989-06-23
2.0	1996-12-31
2.02	1998-04-20
2.03	1999-02-19
2.04	2001-03-21
2.05b	2002-07-17
3.0	2004-08-03
3.1	2005-12-08
3.2	2006-10-11
4.0	2009-02-20
4.1	2009-12-31
4.2	2011-02-13
4.3	2014-02-26
4.4	2016-09-15

Section 1.1: Hello World

Interactive Shell

The Bash shell is commonly used **interactively:** It lets you enter and edit commands, then executes them when you press the `Return` key. Many Unix-based and Unix-like operating systems use Bash as their default shell (notably Linux and macOS). The terminal automatically enters an interactive Bash shell process on startup.

Output `Hello World` by typing the following:

```
echo "Hello World"
#> Hello World  # Output Example
```

Notes

- You can change the shell by just typing the name of the shell in terminal. For example: `sh`, `bash`, etc.
- `echo` is a Bash builtin command that writes the arguments it receives to the standard output. It appends a newline to the output, by default.

Non-Interactive Shell

The Bash shell can also be run **non-interactively** from a script, making the shell require no human interaction. Interactive behavior and scripted behavior should be identical – an important design consideration of Unix V7 Bourne shell and transitively Bash. Therefore anything that can be done at the command line can be put in a script file for reuse.

Follow these steps to create a `Hello World` script:

1. Create a new file called `hello-world.sh`

```
touch hello-world.sh
```

2. Make the script executable by running `chmod +x hello-world.sh`

3. Add this code:

```
#!/bin/bash
echo "Hello World"
```

Line 1: The first line of the script must start with the character sequence `#!`, referred to as *shebang*1. The shebang instructs the operating system to run `/bin/bash`, the Bash shell, passing it the script's path as an argument.

E.g. `/bin/bash hello-world.sh`

Line 2: Uses the `echo` command to write `Hello World` to the standard output.

4. Execute the `hello-world.sh` script from the command line using one of the following:

   - `./hello-world.sh` – most commonly used, and recommended
   - `/bin/bash hello-world.sh`
   - `bash hello-world.sh` – assuming `/bin` is in your `$PATH`
   - `sh hello-world.sh`

For real production use, you would omit the `.sh` extension (which is misleading anyway, since this is a Bash script, not a `sh` script) and perhaps move the file to a directory within your PATH so that it is available to you regardless of your current working directory, just like a system command such as `cat` or `ls`.

Common mistakes include:

1. Forgetting to apply execute permission on the file, i.e., `chmod +x hello-world.sh`, resulting in the output of `./hello-world.sh: Permission denied`.

2. Editing the script on Windows, which produces incorrect line ending characters that Bash cannot handle.

   A common symptom is `: command not found` where the carriage return has forced the cursor to the beginning of line, overwriting the text before the colon in the error message.

   The script can be fixed using the `dos2unix` program.

   An example use: `dos2unix hello-world.sh`

   `dos2unix` edits the file inline.

3. Using `sh ./hello-world.sh`, not realizing that `bash` and `sh` are distinct shells with distinct features (though since Bash is backwards-compatible, the opposite mistake is harmless).

   Anyway, simply relying on the script's shebang line is vastly preferable to explicitly writing `bash` or `sh` (or `python` or `perl` or `awk` or `ruby` or...) before each script's file name.

   A common shebang line to use in order to make your script more portable is to use `#!/usr/bin/env bash` instead of hard-coding a path to Bash. That way, `/usr/bin/env` has to exist, but beyond that point, `bash` just

needs to be on your PATH. On many systems, `/bin/bash` doesn't exist, and you should use `/usr/local/bin/bash` or some other absolute path; this change avoids having to figure out the details of that.

1 *Also referred to as sha-bang, hashbang, pound-bang, hash-pling.*

Section 1.2: Hello World Using Variables

Create a new file called `hello.sh` with the following content and give it executable permissions with `chmod +x hello.sh`.

> Execute/Run via: `./hello.sh`

```
#!/usr/bin/env bash

# Note that spaces cannot be used around the `=` assignment operator
whom_variable="World"

# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

This will print `Hello, World` to standard output when executed.

To tell bash where the script is you need to be very specific, by pointing it to the containing directory, normally with `./` if it is your working directory, where `.` is an alias to the current directory. If you do not specify the directory, `bash` tries to locate the script in one of the directories contained in the `$PATH` environment variable.

The following code accepts an argument $1, which is the first command line argument, and outputs it in a formatted string, following `Hello,`.

> Execute/Run via: `./hello.sh World`

```
#!/usr/bin/env bash
printf "Hello, %s\n" "$1"
#> Hello, World
```

It is important to note that $1 has to be quoted in double quote, not single quote. `"$1"` expands to the first command line argument, as desired, while `'$1'` evaluates to literal string $1.

> **Security Note:**
> Read **Security implications of forgetting to quote a variable in bash shells** to understand the importance of placing the variable text within double quotes.

Section 1.3: Hello World with User Input

The following will prompt a user for input, and then store that input as a string (text) in a variable. The variable is then used to give a message to the user.

```
#!/usr/bin/env bash
echo  "Who are you?"
read name
echo "Hello, $name."
```

The command `read` here reads one line of data from standard input into the variable `name`. This is then referenced using `$name` and printed to standard out using `echo`.

Example output:

```
$ ./hello_world.sh
Who are you?
Matt
Hello, Matt.
```

Here the user entered the name "Matt", and this code was used to say `Hello, Matt.`.

And if you want to append something to the variable value while printing it, use curly brackets around the variable name as shown in the following example:

```
#!/usr/bin/env bash
echo  "What are you doing?"
read action
echo "You are ${action}ing."
```

Example output:

```
$ ./hello_world.sh
What are you doing?
Sleep
You are Sleeping.
```

Here when user enters an action, "ing" is appended to that action while printing.

Section 1.4: Importance of Quoting in Strings

Quoting is important for string expansion in bash. With these, you can control how the bash parses and expands your strings.

There are two types of quoting:

- **Weak**: *uses double quotes: "*
- **Strong**: *uses single quotes: '*

If you want to bash to expand your argument, you can use **Weak Quoting**:

```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```

If you don't want to bash to expand your argument, you can use **Strong Quoting**:

```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```

```
#> Hello $world
```

You can also use escape to prevent expansion:

```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

For more detailed information other than beginner details, you can continue to read it here.

Section 1.5: Viewing information for Bash built-ins

```
help <command>
```

This will display the Bash help (manual) page for the specified built-in.

For example, `help unset` will show:

> ```
> unset: unset [-f] [-v] [-n] [name ...]
> Unset values and attributes of shell variables and functions.
>
> For each NAME, remove the corresponding variable or function.
>
> Options:
> -f    treat each NAME as a shell function
> -v    treat each NAME as a shell variable
> -n    treat each NAME as a name reference and unset the variable itself
> rather than the variable it references
>
> Without options, unset first tries to unset a variable, and if that fails,
> tries to unset a function.
>
> Some variables cannot be unset; also see `readonly'.
>
> Exit Status:
> Returns success unless an invalid option is given or a NAME is read-only.
> ```

To see a list of all built-ins with a short description, use

```
help -d
```

Section 1.6: Hello World in "Debug" mode

```
$ cat hello.sh
#!/bin/bash
echo "Hello World"
$ bash -x hello.sh
+ echo Hello World
Hello World
```

The `-x` argument enables you to walk through each line in the script. One good example is here:

```
$ cat hello.sh
#!/bin/bash
echo "Hello World\n"
```

```
adding_string_to_number="s"
v=$(expr 5 + $adding_string_to_number)

$ ./hello.sh
Hello World

expr: non-integer argument
```

The above prompted error is not enough to trace the script; however, using the following way gives you a better sense where to look for the error in the script.

```
$ bash -x hello.sh
+ echo Hello World\n
Hello World

+ adding_string_to_number=s
+ expr 5 + s
expr: non-integer argument
+ v=
```

Section 1.7: Handling Named Arguments

```
#!/bin/bash

deploy=false
uglify=false

while (( $# > 1 )); do case $1 in
   --deploy) deploy="$2";;
   --uglify) uglify="$2";;
   *) break;
 esac; shift 2
done

$deploy && echo "will deploy... deploy = $deploy"
$uglify && echo "will uglify... uglify = $uglify"

# how to run
# chmod +x script.sh
# ./script.sh --deploy true --uglify false
```

Chapter 2: Script shebang

Section 2.1: Env shebang

To execute a script file with the **bash** executable found in the PATH environment variable by using the executable **env**, the **first line** of a script file must indicate the absolute path to the **env** executable with the argument **bash**:

```
#!/usr/bin/env bash
```

The **env** path in the shebang is resolved and used only if a script is directly launch like this:

```
script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.2: Direct shebang

To execute a script file with the **bash** interpreter, the **first line** of a script file must indicate the absolute path to the **bash** executable to use:

```
#!/bin/bash
```

The **bash** path in the shebang is resolved and used only if a script is directly launch like this:

```
./script.sh
```

The script must have execution permission.

The shebang is ignored when a **bash** interpreter is explicitly indicated to execute a script:

```
bash script.sh
```

Section 2.3: Other shebangs

There are two kinds of programs the kernel knows of. A binary program is identified by it's ELF (**E**xtenable**L**oadable**F**ormat) header, which is usually produced by a compiler. The second one are scripts of any kind.

If a file starts in the very first line with the sequence **#!** then the next string has to be a pathname of an interpreter. If the kernel reads this line, it calls the interpreter named by this pathname and gives all of the following words in this line as arguments to the interpreter. If there is no file named "something" or "wrong":

```
#!/bin/bash something wrong
echo "This line never gets printed"
```

bash tries to execute its argument "something wrong" which doesn't exist. The name of the script file is added too. To see this clearly use an **echo** shebang:

```
#"/bin/echo something wrong
# and now call this script named "thisscript" like so:
# thisscript one two
# the output will be:
something wrong ./thisscript one two
```

Some programs like **awk** use this technique to run longer scripts residing in a disk file.

Chapter 3: Navigating directories

Section 3.1: Absolute vs relative directories

To change to an absolutely specified directory, use the entire name, starting with a slash `/`, thus:

```
cd /home/username/project/abc
```

If you want to change to a directory near your current on, you can specify a relative location. For example, if you are already in `/home/username/project`, you can enter the subdirectory `abc` thus:

```
cd abc
```

If you want to go to the directory above the current directory, you can use the alias `..`. For example, if you were in `/home/username/project/abc` and wanted to go to `/home/username/project`, then you would do the following:

```
cd ..
```

This may also be called going "up" a directory.

Section 3.2: Change to the last directory

For the current shell, this takes you to the previous directory that you were in, no matter where it was.

```
cd -
```

Doing it multiple times effectively "toggles" you being in the current directory or the previous one.

Section 3.3: Change to the home directory

The default directory is the home directory (`$HOME`, typically `/home/username`), so `cd` without any directory takes you there

```
cd
```

Or you could be more explicit:

```
cd $HOME
```

A shortcut for the home directory is `~`, so that could be used as well.

```
cd ~
```

Section 3.4: Change to the Directory of the Script

In general, there are two types of Bash **scripts**:

1. System tools which operate from the current working directory
2. Project tools which modify files relative to their own place in the files system

For the second type of scripts, it is useful to change to the directory where the script is stored. This can be done with the following command:

```
cd "$(dirname "$(readlink -f "$0")")"
```

This command runs 3 commands:

1. `readlink -f "$0"` determines the path to the current script (`$0`)
2. `dirname` converts the path to script to the path to its directory
3. `cd` changes the current work directory to the directory it receives from `dirname`

Chapter 4: Listing Files

Option	Description
`-a`, `--all`	List all entries including ones that start with a dot
`-A`, `--almost-all`	List all entries excluding `.` and `..`
`-c`	Sort files by change time
`-d`, `--directory`	List directory entries
`-h`, `--human-readable`	Show sizes in human readable format (i.e. `K`, `M`)
`-H`	Same as above only with powers of 1000 instead of 1024
`-l`	Show contents in long-listing format
`-o`	Long -listing format without group info
`-r`, `--reverse`	Show contents in reverse order
`-s`, `--size`	Print size of each file in blocks
`-S`	Sort by file size
`--sort=WORD`	Sort contents by a word. (i.e size, version, status)
`-t`	Sort by modification time
`-u`	Sort by last access time
`-v`	Sort by version
`-1`	List one file per line

Section 4.1: List Files in a Long Listing Format

The `ls` command's `-l` option prints a specified directory's contents in a long listing format. If no directory is specified then, by default, the contents of the current directory are listed.

```
ls -l /etc
```

Example Output:

```
total 1204
drwxr-xr-x  3 root root    4096 Apr 21 03:44 acpi
-rw-r--r--  1 root root    3028 Apr 21 03:38 adduser.conf
drwxr-xr-x  2 root root    4096 Jun 11 20:42 alternatives
...
```

The output first displays `total`, which indicates the total size in **blocks** of all the files in the listed directory. It then displays eight columns of information for each file in the listed directory. Below are the details for each column in the output:

Column No.	Example	Description
1.1	`d`	File type (see table below)
1.2	`rwxr-xr-x`	Permission string
2	`3`	Number of hard links
3	`root`	Owner name
4	`root`	Owner group
5	`4096`	File size in bytes
6	`Apr 21 03:44`	Modification time
7	`acpi`	File name

File Type

The file type can be one of any of the following characters.

Character	File Type
`-`	Regular file
`b`	Block special file
`c`	Character special file
`C`	High performance ("contiguous data") file
`d`	Directory
`D`	Door (special IPC file in Solaris 2.5+ only)
`l`	Symbolic link
`M`	Off-line ("migrated") file (Cray DMF)
`n`	Network special file (HP-UX)
`p`	FIFO (named pipe)
`P`	Port (special system file in Solaris 10+ only)
`s`	Socket
`?`	Some other file type

Section 4.2: List the Ten Most Recently Modified Files

The following will list up to ten of the most recently modified files in the current directory, using a long listing format (`-l`) and sorted by time (`-t`).

```
ls -lt | head
```

Section 4.3: List All Files Including Dotfiles

A **dotfile** is a file whose names begin with a `.`. These are normally hidden by `ls` and not listed unless requested.

For example the following output of `ls`:

```
$ ls
bin  pki
```

The `-a` or `--all` option will list all files, including dotfiles.

```
$ ls -a
.   .ansible       .bash_logout   .bashrc   .lesshst   .puppetlabs   .viminfo
..  .bash_history  .bash_profile  bin       pki        .ssh
```

The `-A` or `--almost-all` option will list all files, including dotfiles, but does not list implied `.` and `..`. Note that `.` is the current directory and `..` is the parent directory.

```
$ ls -A
.ansible       .bash_logout   .bashrc   .lesshst   .puppetlabs   .viminfo
.bash_history  .bash_profile  bin       pki        .ssh
```

Section 4.4: List Files Without Using `ls`

Use the Bash shell's filename expansion and brace expansion capabilities to obtain the filenames:

```
# display the files and directories that are in the current directory
printf "%s\n" *

# display only the directories in the current directory
printf "%s\n" */

# display only (some) image files
printf "%s\n" *.{gif,jpg,png}
```

To capture a list of files into a variable for processing, it is typically good practice to use a bash array:

```
files=( * )

# iterate over them
for file in "${files[@]}"; do
    echo "$file"
done
```

Section 4.5: List Files

The `ls` command lists the contents of a specified directory, **excluding** dotfiles. If no directory is specified then, by default, the contents of the current directory are listed.

Listed files are sorted alphabetically, by default, and aligned in columns if they don't fit on one line.

```
$ ls
apt   configs   Documents  Fonts    Music      Programming  Templates  workspace
bin   Desktop   eclipse    git      Pictures   Public       Videos
```

Section 4.6: List Files in a Tree-Like Format

The `tree` command lists the contents of a specified directory in a tree-like format. If no directory is specified then, by default, the contents of the current directory are listed.

Example Output:

```
$ tree /tmp
/tmp
├── 5037
├── adb.log
└── evince-20965
    └── image.FPWTJY.png
```

Use the `tree` command's `-L` option to limit the display depth and the `-d` option to only list directories.

Example Output:

```
$ tree -L 1 -d /tmp
/tmp
└── evince-20965
```

Section 4.7: List Files Sorted by Size

The `ls` command's `-S` option sorts the files in descending order of file size.

```
$ ls -l -S ./Fruits
total 444
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
```

When used with the `-r` option the sort order is reversed.

```
$ ls -l -S -r /Fruits
total 444
-rw-rw-rw- 1 root root  50197 Jul 28 19:19 bananas.jpg
-rw-rw-rw- 1 root root 102283 Jul 28 19:19 kiwis.jpg
-rw-rw-rw- 1 root root 295303 Jul 28 19:19 apples.jpg
```

Chapter 5: Using cat

Option	Details
-n	Print line numbers
-v	Show non-printing characters using ^ and M- notation except LFD and TAB
-T	Show TAB characters as ^I
-E	Show linefeed(LF) characters as $
-e	Same as -vE
-b	Number nonempty output lines, overrides -n
-A	equivalent to -vET
-s	suppress repeated empty output lines, s refers to squeeze

Section 5.1: Concatenate files

This is the primary purpose of `cat`.

```
cat file1 file2 file3 > file_all
```

`cat` can also be used similarly to concatenate files as part of a pipeline, e.g.

```
cat file1 file2 file3 | grep foo
```

Section 5.2: Printing the Contents of a File

```
cat file.txt
```

will print the contents of a file.

If the file contains non-ASCII characters, you can display those characters symbolically with `cat -v`. This can be quite useful for situations where control characters would otherwise be invisible.

```
cat -v unicode.txt
```

Very often, for interactive use, you are better off using an interactive pager like `less` or `more`, though. (`less` is far more powerful than `more` and it is advised to use `less` more often than `more`.)

```
less file.txt
```

To pass the contents of a file as input to a command. An approach usually seen as better (UUOC) is to use redirection.

```
tr A-Z a-z <file.txt   # as an alternative to cat file.txt | tr A-Z a-z
```

In case the content needs to be listed backwards from its end the command `tac` can be used:

```
tac file.txt
```

If you want to print the contents with line numbers, then use `-n` with `cat`:

```
cat -n file.txt
```

To display the contents of a file in a completely unambiguous byte-by-byte form, a hex dump is the standard solution. This is good for very brief snippets of a file, such as when you don't know the precise encoding. The standard hex dump utility is `od -cH`, though the representation is slightly cumbersome; common replacements include `xxd` and `hexdump`.

```
$ printf 'Hëllö wörld' | xxd
0000000: 48c3 ab6c 6cc3 b620 77c3 b672 6c64       H..ll.. w..rld
```

Section 5.3: Write to a file

```
cat >file
```

It will let you write the text on terminal which will be saved in a file named *file*.

```
cat >>file
```

will do the same, except it will append the text to the end of the file.

N.B: `Ctrl+D` to end writing text on terminal (Linux)

A here document can be used to inline the contents of a file into a command line or a script:

```
cat <<END >file
Hello, World.
END
```

The token after the `<<` redirection symbol is an arbitrary string which needs to occur alone on a line (with no leading or trailing whitespace) to indicate the end of the here document. You can add quoting to prevent the shell from performing command substitution and variable interpolation:

```
cat <<'fnord'
Nothing in `here` will be $changed
fnord
```

(Without the quotes, `here` would be executed as a command, and `$changed` would be substituted with the value of the variable `changed` -- or nothing, if it was undefined.)

Section 5.4: Show non printable characters

This is useful to see if there are any non-printable characters, or non-ASCII characters.

e.g. If you have copy-pasted the code from web, you may have quotes like `”` instead of standard `"`.

```
$ cat -v file.txt
$ cat -vE file.txt # Useful in detecting trailing spaces.
```

e.g.

```
$ echo '”      ' | cat -vE # echo | will be replaced by actual file.
M-bM-^@M-^]      $
```

You may also want to use `cat -A` (A for All) that is equivalent to `cat -vET`. It will display TAB characters (displayed

as `^I`), non printable characters and end of each line:

```
$ echo '" `' | cat -A
M-bM-^@M-^]^I`$
```

Section 5.5: Read from standard input

```
cat < file.txt
```

Output is same as `cat file.txt`, but it reads the contents of the file from standard input instead of directly from the file.

```
printf "first line\nSecond line\n" | cat -n
```

The echo command before `|` outputs two lines. The cat command acts on the output to add line numbers.

Section 5.6: Display line numbers with output

Use the `--number` flag to print line numbers before each line. Alternatively, `-n` does the same thing.

```
$ cat --number file

 1  line 1
 2  line 2
 3
 4  line 4
 5  line 5
```

To skip empty lines when counting lines, use the `--number-nonblank`, or simply `-b`.

```
$ cat -b file

 1  line 1
 2  line 2

 3  line 4
 4  line 5
```

Section 5.7: Concatenate gzipped files

Files compressed by `gzip` can be directly concatenated into larger gzipped files.

```
cat file1.gz file2.gz file3.gz > combined.gz
```

This is a property of `gzip` that is less efficient than concatenating the input files and gzipping the result:

```
cat file1 file2 file3 | gzip > combined.gz
```

A complete demonstration:

```
echo 'Hello world!' > hello.txt
echo 'Howdy world!' > howdy.txt
gzip hello.txt
gzip howdy.txt
```

```
cat hello.txt.gz howdy.txt.gz > greetings.txt.gz

gunzip greetings.txt.gz

cat greetings.txt
```

Which results in

```
Hello world!
Howdy world!
```

Notice that `greetings.txt.gz` is a ***single file*** and is decompressed as the ***single file*** `greeting.txt`. Contrast this with `tar -czf hello.txt howdy.txt > greetings.tar.gz`, which keeps the files separate inside the tarball.

Chapter 6: Grep

Section 6.1: How to search a file for a pattern

To find the word **foo** in the file *bar* :

```
grep foo ~/Desktop/bar
```

To find all lines that **do not** contain foo in the file *bar* :

```
grep -v foo ~/Desktop/bar
```

To use find all words containing foo in the end (WIldcard Expansion):

```
grep "*foo" ~/Desktop/bar
```

Chapter 7: Aliasing

Shell aliases are a simple way to create new commands or to wrap existing commands with code of your own. They somewhat overlap with shell functions, which are however more versatile and should therefore often be preferred.

Section 7.1: Bypass an alias

Sometimes you may want to bypass an alias temporarily, without disabling it. To work with a concrete example, consider this alias:

```
alias ls='ls --color=auto'
```

And let's say you want to use the `ls` command without disabling the alias. You have several options:

- Use the `command` builtin: `command ls`
- Use the full path of the command: `/bin/ls`
- Add a `\` anywhere in the command name, for example: `\ls`, or `l\s`
- Quote the command: `"ls"` or `'ls'`

Section 7.2: Create an Alias

```
alias word='command'
```

Invoking `word` will run `command`. Any arguments supplied to the alias are simply appended to the target of the alias:

```
alias myAlias='some command --with --options'
myAlias foo bar baz
```

The shell will then execute:

```
some command --with --options foo bar baz
```

To include multiple commands in the same alias, you can string them together with **&&**. For example:

```
alias print_things='echo "foo" && echo "bar" && echo "baz"'
```

Section 7.3: Remove an alias

To remove an existing alias, use:

```
unalias {alias_name}
```

Example:

```
# create an alias
$ alias now='date'

# preview the alias
$ now
Thu Jul 21 17:11:25 CEST 2016

# remove the alias
$ unalias now
```

```
# test if removed
$ now
-bash: now: command not found
```

Section 7.4: The BASH_ALIASES is an internal bash assoc array

Aliases are named shortcuts of commands, one can define and use in interactive bash instances. They are held in an associative array named BASH_ALIASES. To use this var in a script, it must be run within an interactive shell

```
#!/bin/bash -li
# note the -li above! -l makes this behave like a login shell
# -i makes it behave like an interactive shell
#
# shopt -s expand_aliases will not work in most cases

echo There are ${#BASH_ALIASES[*]} aliases defined.

for ali in "${!BASH_ALIASES[@]}"; do
   printf "alias: %-10s triggers: %s\n" "$ali" "${BASH_ALIASES[$ali]}"
done
```

Section 7.5: Expand alias

Assuming that `bar` is an alias for `someCommand -flag1`.

Type `bar` on the command line and then press `Ctrl` + `alt` + `e`

you'll get `someCommand -flag1` where `bar` was standing.

Section 7.6: List all Aliases

```
alias -p
```

will list all the current aliases.

Chapter 8: Jobs and Processes

Section 8.1: Job handling

Creating jobs

To create an job, just append a single `&` after the command:

```
$ sleep 10 &
[1] 20024
```

You can also make a running process a job by pressing `Ctrl` + `z`:

```
$ sleep 10
^Z
[1]+  Stopped                 sleep 10
```

Background and foreground a process

To bring the Process to the foreground, the command `fg` is used together with `%`

```
$ sleep 10 &
[1] 20024

$ fg %1
sleep 10
```

Now you can interact with the process. To bring it back to the background you can use the `bg` command. Due to the occupied terminal session, you need to stop the process first by pressing `Ctrl` + `z`.

```
$ sleep 10
^Z
[1]+  Stopped              sleep 10

$ bg %1
[1]+ sleep 10 &
```

Due to the laziness of some Programmers, all these commands also work with a single `%` if there is only one process, or for the first process in the list. For Example:

```
$ sleep 10 &
[1] 20024

$ fg %        # to bring a process to foreground 'fg %' is also working.
sleep 10
```

or just

```
$ %           # laziness knows no boundaries, '%' is also working.
sleep 10
```

Additionally, just typing `fg` or `bg` without any argument handles the last job:

```
$ sleep 20 &
$ sleep 10 &
$ fg
```

```
sleep 10
^C
$ fg
sleep 20
```

Killing running jobs

```
$ sleep 10 &
[1] 20024

$ kill %1
[1]+  Terminated                    sleep 10
```

The sleep process runs in the background with process id (pid) `20024` and job number `1`. In order to reference the process, you can use either the pid or the job number. If you use the job number, you must prefix it with `%`. The default kill signal sent by `kill` is `SIGTERM`, which allows the target process to exit gracefully.

Some common kill signals are shown below. To see a full list, run `kill -l`.

Signal name	Signal value	Effect
SIGHUP	1	Hangup
SIGINT	2	Interrupt from keyboard
SIGKILL	9	Kill signal
SIGTERM	15	Termination signal

Start and kill specific processes

Probably the easiest way of killing a running process is by selecting it through the process name as in the following example using `pkill` command as

```
pkill -f test.py
```

(or) a more fool-proof way using `pgrep` to search for the actual process-id

```
kill $(pgrep -f 'python test.py')
```

The same result can be obtained using `grep` over `ps -ef | grep name_of_process` then killing the process associated with the resulting pid (process id). Selecting a process using its name is convinient in a testing environment but can be really dangerous when the script is used in production: it is virtually impossible to be sure that the name will match the process you actually want to kill. In those cases, the following approach is actually much safe.

Start the script that will eventually killed with the following approach. Let's assume that the command you want to execute and eventually kill is `python test.py`.

```
#!/bin/bash

if [[ ! -e /tmp/test.py.pid ]]; then   # Check if the file already exists
    python test.py &                   #+and if so do not run another process.
    echo $! > /tmp/test.py.pid
else
    echo -n "ERROR: The process is already running with pid "
    cat /tmp/test.py.pid
    echo
fi
```

This will create a file in the `/tmp` directory containing the pid of the `python test.py` process. If the file already exists, we assume that the command is already running and the script return an error.

Then, when you want to kill it use the following script:

```
#!/bin/bash

if [[ -e /tmp/test.py.pid ]]; then   # If the file do not exists, then the
    kill `cat /tmp/test.py.pid`      #+the process is not running. Useless
    rm /tmp/test.py.pid              #+trying to kill it.
else
    echo "test.py is not running"
fi
```

that will kill exactly the process associated with your command, without relying on any volatile information (like the string used to run the command). Even in this case if the file does not exist, the script assume that you want to kill a non-running process.

This last example can be easily improved for running the same command multiple times (appending to the pid file instead of overwriting it, for example) and to manage cases where the process dies before being killed.

Section 8.2: Check which process running on specific port

To check which process running on port 8080

```
lsof -i :8080
```

Section 8.3: Disowning background job

```
$ gzip extremelylargefile.txt &
$ bg
$ disown %1
```

This allows a long running process to continue once your shell (terminal, ssh, etc) is closed.

Section 8.4: List Current Jobs

```
$ tail -f /var/log/syslog > log.txt
[1]+  Stopped                 tail -f /var/log/syslog > log.txt

$ sleep 10 &

$ jobs
[1]+  Stopped                 tail -f /var/log/syslog > log.txt
[2]-  Running                 sleep 10 &
```

Section 8.5: Finding information about a running process

`ps aux | grep <search-term>` shows processes matching *search-term*

Example:

```
root@server7:~# ps aux | grep nginx
root       315  0.0  0.3 144392  1020 ?        Ss   May28   0:00 nginx: master process
/usr/sbin/nginx
www-data  5647  0.0  1.1 145124  3048 ?        S    Jul18   2:53 nginx: worker process
www-data  5648  0.0  0.1 144392   376 ?        S    Jul18   0:00 nginx: cache manager process
root     13134  0.0  0.3   4960   920 pts/0    S+   14:33   0:00 grep --color=auto nginx
root@server7:~#
```

Here, second column is the process id. For example, if you want to kill the nginx process, you can use the command `kill 5647`. It is always adviced to use the `kill` command with `SIGTERM` rather than `SIGKILL`.

Section 8.6: List all processes

There are two common ways to list all processes on a system. Both list all processes running by all users, though they differ in the format they output (the reason for the differences are historical).

```
ps -ef    # lists all processes
ps aux    # lists all processes in alternative format (BSD)
```

This can be used to check if a given application is running. For example, to check if the SSH server (sshd) is running:

```
ps -ef | grep sshd
```

Chapter 9: Redirection

Parameter	Details
internal file descriptor	An integer.
direction	One of `>`, `<` or `<>`
external file descriptor or path	`&` followed by an integer for file descriptor or a path.

Section 9.1: Redirecting standard output

`>` redirect the standard output (aka `STDOUT`) of the current command into a file or another descriptor.

These examples write the output of the `ls` command into the file `file.txt`

```
ls >file.txt
> file.txt ls
```

The target file is created if it doesn't exists, otherwise this file is truncated.

The default redirection descriptor is the standard output or `1` when none is specified. This command is equivalent to the previous examples with the standard output explicitly indicated:

```
ls 1>file.txt
```

Note: the redirection is initialized by the executed shell and not by the executed command, therefore it is done before the command execution.

Section 9.2: Append vs Truncate

Truncate `>`

1. Create specified file if it does not exist.
2. Truncate (remove file's content)
3. Write to file

```
$ echo "first line" > /tmp/lines
$ echo "second line" > /tmp/lines

$ cat /tmp/lines
second line
```

Append `>>`

1. Create specified file if it does not exist.
2. Append file (writing at end of file).

```
# Overwrite existing file
$ echo "first line" > /tmp/lines

# Append a second line
$ echo "second line" >> /tmp/lines

$ cat /tmp/lines
first line
second line
```

Section 9.3: Redirecting both STDOUT and STDERR

File descriptors like `0` and `1` are pointers. We change what file descriptors point to with redirection. `>/dev/null` means `1` points to `/dev/null`.

First we point `1` (`STDOUT`) to `/dev/null` then point `2` (`STDERR`) to whatever `1` points to.

```
# STDERR is redirect to STDOUT: redirected to /dev/null,
# effectually redirecting both STDERR and STDOUT to /dev/null
echo 'hello' > /dev/null 2>&1
```

Version ≥ 4.0

This *can* be further shortened to the following:

```
echo 'hello' &> /dev/null
```

However, this form may be undesirable in production if shell compatibility is a concern as it conflicts with POSIX, introduces parsing ambiguity, and shells without this feature will misinterpret it:

```
# Actual code
echo 'hello' &> /dev/null
echo 'hello' &> /dev/null 'goodbye'

# Desired behavior
echo 'hello' > /dev/null 2>&1
echo 'hello' 'goodbye' > /dev/null 2>&1

# Actual behavior
echo 'hello' &
echo 'hello' & goodbye > /dev/null
```

NOTE: `&>` is known to work as desired in both Bash and Zsh.

Section 9.4: Using named pipes

Sometimes you may want to output something by one program and input it into another program, but can't use a standard pipe.

```
ls -l | grep ".log"
```

You could simply write to a temporary file:

```
touch tempFile.txt
ls -l > tempFile.txt
grep ".log" < tempFile.txt
```

This works fine for most applications, however, nobody will know what `tempFile` does and someone might remove it if it contains the output of `ls -l` in that directory. This is where a named pipe comes into play:

```
mkfifo myPipe
ls -l > myPipe
grep ".log" < myPipe
```

`myPipe` is technically a file (everything is in Linux), so let's do `ls -l` in an empty directory that we just created a pipe in:

```
mkdir pipeFolder
cd pipeFolder
mkfifo myPipe
ls -l
```

The output is:

```
prw-r--r-- 1 root root 0 Jul 25 11:20 myPipe
```

Notice the first character in the permissions, it's listed as a pipe, not a file.

Now let's do something cool.

Open one terminal, and make note of the directory (or create one so that cleanup is easy), and make a pipe.

```
mkfifo myPipe
```

Now let's put something in the pipe.

```
echo "Hello from the other side" > myPipe
```

You'll notice this hangs, the other side of the pipe is still closed. Let's open up the other side of the pipe and let that stuff through.

Open another terminal and go to the directory that the pipe is in (or if you know it, prepend it to the pipe):

```
cat < myPipe
```

You'll notice that after `hello from the other side` is output, the program in the first terminal finishes, as does that in the second terminal.

Now run the commands in reverse. Start with `cat < myPipe` and then echo something into it. It still works, because a program will wait until something is put into the pipe before terminating, because it knows it has to get something.

Named pipes can be useful for moving information between terminals or between programs.

Pipes are small. Once full, the writer blocks until some reader reads the contents, so you need to either run the reader and writer in different terminals or run one or the other in the background:

```
 ls -l /tmp > myPipe &
 cat < myPipe
```

More examples using named pipes:

- Example 1 - all commands on the same terminal / same shell

```
$ { ls -l && cat file3; } >mypipe &
$ cat <mypipe
# Output: Prints ls -l data and then prints file3 contents on screen
```

- Example 2 - all commands on the same terminal / same shell

```
$ ls -l >mypipe &
$ cat file3 >mypipe &
```

```
$ cat <mypipe
#Output: This prints on screen the contents of mypipe.
```

Mind that first contents of `file3` are displayed and then the `ls -l` data is displayed (LIFO configuration).

- Example 3 - all commands on the same terminal / same shell

```
$ { pipedata=$(<mypipe) && echo "$pipedata"; } &
$ ls >mypipe
# Output: Prints the output of ls directly on screen
```

Mind that the variable `$pipedata` is not available for usage in the main terminal / main shell since the use of `&` invokes a subshell and `$pipedata` was only available in this subshell.

- Example 4 - all commands on the same terminal / same shell

```
$ export pipedata
$ pipedata=$(<mypipe) &
$ ls -l *.sh >mypipe
$ echo "$pipedata"
#Output : Prints correctly the contents of mypipe
```

This prints correctly the value of `$pipedata` variable in the main shell due to the export declaration of the variable. The main terminal/main shell is not hanging due to the invocation of a background shell (`&`).

Section 9.5: Redirection to network addresses

`Version ≥ 2.04`

Bash treats some paths as special and can do some network communication by writing to `/dev/{udp|tcp}/host/port`. Bash cannot setup a listening server, but can initiate a connection, and for TCP can read the results at least.

For example, to send a simple web request one could do:

```
exec 3</dev/tcp/www.google.com/80
printf 'GET / HTTP/1.0\r\n\r\n' >&3
cat <&3
```

and the results of `www.google.com`'s default web page will be printed to `stdout`.

Similarly

```
printf 'HI\n' >/dev/udp/192.168.1.1/6666
```

would send a UDP message containing `HI\n` to a listener on `192.168.1.1:6666`

Section 9.6: Print error messages to stderr

Error messages are generally included in a script for debugging purposes or for providing rich user experience. Simply writing error message like this:

```
cmd || echo 'cmd failed'
```

may work for simple cases but it's not the usual way. In this example, the error message will pollute the actual output of the script by mixing both errors and successful output in `stdout`.

In short, error message should go to `stderr` not `stdout`. It's pretty simple:

```
cmd || echo 'cmd failed' >/dev/stderr
```

Another example:

```
if cmd; then
    echo 'success'
else
    echo 'cmd failed' >/dev/stderr
fi
```

In the above example, the success message will be printed on `stdout` while the error message will be printed on `stderr`.

A better way to print error message is to define a function:

```
err(){
    echo "E: $*" >>/dev/stderr
}
```

Now, when you have to print an error:

```
err "My error message"
```

Section 9.7: Redirecting multiple commands to the same file

```
{
  echo "contents of home directory"
  ls ~
} > output.txt
```

Section 9.8: Redirecting STDIN

`<` reads from its right argument and writes to its left argument.

To write a file into `STDIN` we should *read* `/tmp/a_file` and *write* into `STDIN` i.e `0</tmp/a_file`

Note: Internal file descriptor defaults to `0` (`STDIN`) for `<`

```
$ echo "b" > /tmp/list.txt
$ echo "a" >> /tmp/list.txt
$ echo "c" >> /tmp/list.txt
$ sort < /tmp/list.txt
a
b
c
```

Section 9.9: Redirecting STDERR

2 is STDERR.

```
$ echo_to_stderr 2>/dev/null # echos nothing
```

Definitions:

`echo_to_stderr` is a command that writes `"stderr"` to STDERR

```
echo_to_stderr () {
    echo stderr >&2
}

$ echo_to_stderr
stderr
```

Section 9.10: STDIN, STDOUT and STDERR explained

Commands have one input (STDIN) and two kinds of outputs, standard output (STDOUT) and standard error (STDERR).

For example:

STDIN

```
root@server~# read
Type some text here
```

Standard input is used to provide input to a program. (Here we're using the `read` builtin to read a line from STDIN.)

STDOUT

```
root@server~# ls file
file
```

Standard output is generally used for "normal" output from a command. For example, `ls` lists files, so the files are sent to STDOUT.

STDERR

```
root@server~# ls anotherfile
ls: cannot access 'anotherfile': No such file or directory
```

Standard error is (as the name implies) used for error messages. Because this message is not a list of files, it is sent to STDERR.

STDIN, STDOUT and STDERR are the three *standard streams.* They are identified to the shell by a number rather than a name:

0 = Standard in
1 = Standard out
2 = Standard error

By default, STDIN is attached to the keyboard, and both STDOUT and STDERR appear in the terminal. However, we

can redirect either STDOUT or STDERR to whatever we need. For example, let's say that you only need the standard out and all error messages printed on standard error should be suppressed. That's when we use the descriptors `1` and `2`.

Redirecting STDERR to /dev/null

Taking the previous example,

```
root@server~# ls anotherfile 2>/dev/null
root@server~#
```

In this case, if there is any STDERR, it will be redirected to /dev/null (a special file which ignores anything put into it), so you won't get any error output on the shell.

Chapter 10: Control Structures

Parameter to [or test	Details
File Operators	Details
`-e "$file"`	Returns true if the file exists.
`-d "$file"`	Returns true if the file exists and is a directory
`-f "$file"`	Returns true if the file exists and is a regular file
`-h "$file"`	Returns true if the file exists and is a symbolic link
String Comparators	Details
`-z "$str"`	True if length of string is zero
`-n "$str`	True if length of string is non-zero
`"$str" = "$str2"`	True if string $str is equal to string $str2. Not best for integers. It may work but will be inconsitent
`"$str" != "$str2"`	True if the strings are not equal
Integer Comparators	Details
`"$int1" -eq "$int2"`	True if the integers are equal
`"$int1" -ne "$int2"`	True if the integers are not equals
`"$int1" -gt "$int2"`	True if int1 is greater than int 2
`"$int1" -ge "$int2"`	True if int1 is greater than or equal to int2
`"$int1" -lt "$int2"`	True if int1 is less than int 2
`"$int1" -le "$int2"`	True if int1 is less than or equal to int2

Section 10.1: Conditional execution of command lists

How to use conditional execution of command lists

Any builtin command, expression, or function, as well as any external command or script can be executed conditionally using the `&&`*(and)* and `||`*(or)* operators.

For example, this will only print the current directory if the `cd` command was successful.

```
cd my_directory && pwd
```

Likewise, this will exit if the `cd` command fails, preventing catastrophe:

```
cd my_directory || exit
rm -rf *
```

When combining multiple statements in this manner, it's important to remember that (unlike many C-style languages) these operators have no precedence and are left-associative.

Thus, this statement will work as expected...

```
cd my_directory && pwd || echo "No such directory"
```

- If the `cd` succeeds, the `&& pwd` executes and the current working directory name is printed. Unless `pwd` fails (a rarity) the `|| echo ...` will not be executed.
- If the `cd` fails, the `&& pwd` will be skipped and the `|| echo ...` will run.

But this will not (if you're thinking `if...then...else`)...

```
cd my_directory && ls || echo "No such directory"
```

- If the cd fails, the `&& ls` is skipped and the `|| echo ...` is executed.
- If the cd succeeds, the `&& ls` is executed.
  - If the ls succeeds, the `|| echo ...` is ignored. *(so far so good)*
  - ***BUT... if the ls fails, the `|| echo ...` will also be executed.***

> ***It is the ls***, not the cd, ***that is the previous command***.

Why use conditional execution of command lists

Conditional execution is a hair faster than `if...then` but its main advantage is allowing functions and scripts to exit early, or "short circuit".

Unlike many languages like `C` where memory is explicitly allocated for structs and variables and such (and thus must be deallocated), **bash** handles this under the covers. In most cases, we don't have to clean up anything before leaving the function. A **return** statement will deallocate everything local to the function and pickup execution at the return address on the stack.

Returning from functions or exiting scripts as soon as possible can thus significantly improve performance and reduce system load by avoiding the unnecessary execution of code. For example...

```
my_function () {

    ### ALWAYS CHECK THE RETURN CODE

    # one argument required. "" evaluates to false(1)
    [[ "$1" ]]              || return 1

    # work with the argument. exit on failure
    do_something_with "$1" || return 1
    do_something_else      || return 1

    # Success! no failures detected, or we wouldn't be here
    return 0
}
```

Section 10.2: If statement

```
if [[ $1 -eq 1 ]]; then
    echo "1 was passed in the first parameter"
elif [[ $1 -gt 2 ]]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

The closing `fi` is necessary, but the **elif** and/or the **else** clauses can be omitted.

The semicolons before **then** are standard syntax for combining two commands on a single line; they can be omitted only if **then** is moved to the next line.

It's important to understand that the brackets `[[` are not part of the syntax, but are treated as a command; it is the exit code from this command that is being tested. Therefore, you must always include spaces around the brackets.

This also means that the result of any command can be tested. If the exit code from the command is a zero, the statement is considered true.

```
if grep "foo" bar.txt; then
    echo "foo was found"
else
    echo "foo was not found"
fi
```

Mathematical expressions, when placed inside double parentheses, also return 0 or 1 in the same way, and can also be tested:

```
if (( $1 + 5 > 91 )); then
    echo "$1 is greater than 86"
fi
```

You may also come across `if` statements with single brackets. These are defined in the POSIX standard and are guaranteed to work in all POSIX-compliant shells including Bash. The syntax is very similar to that in Bash:

```
if [ "$1" -eq 1 ]; then
    echo "1 was passed in the first parameter"
elif [ "$1" -gt 2 ]; then
    echo "2 was not passed in the first parameter"
else
    echo "The first parameter was not 1 and is not more than 2."
fi
```

Section 10.3: Looping over an array

`for` loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
done
```

Or

```
for ((i=0;i<${#arr[@]};i++));do
    echo "${arr[$i]}"
done
```

`while` loop:

```
i=0
while [ $i -lt ${#arr[@]} ];do
    echo "${arr[$i]}"
    i=$(expr $i + 1)
done
```

Or

```
i=0
while (( $i < ${#arr[@]} ));do
    echo "${arr[$i]}"
    ((i++))
```

```
done
```

Section 10.4: Using For Loop to List Iterate Over Numbers

```
#! /bin/bash

for i in {1..10}; do # {1..10} expands to "1 2 3 4 5 6 7 8 9 10"
    echo $i
done
```

This outputs the following:

```
1
2
3
4
5
6
7
8
8
10
```

Section 10.5: continue and break

Example for continue

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

Example for break

```
for i in [series]
do
    command 4
    if (condition) # Condition to break the loop
    then
            command 5 # Command if the loop needs to be broken
            break
    fi
    command 6 # Command to run if the "condition" is never true
done
```

Section 10.6: Loop break

Break multiple loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
```

```
    for j in "${arr[@]}";do
        echo "$j"
        break 2
    done
done
```

Output:

```
a
a
```

Break single loop:

```
arr=(a b c d e f)
for i in "${arr[@]}";do
    echo "$i"
    for j in "${arr[@]}";do
        echo "$j"
        break
    done
done
```

Output:

```
a
a
b
a
c
a
d
a
e
a
f
a
```

Section 10.7: While Loop

```
#! /bin/bash

i=0

while [ $i -lt 5 ] #While i is less than 5
do
    echo "i is currently $i"
    i=$[$i+1] #Not the lack of spaces around the brackets. This makes it a not a test expression
done #ends the loop
```

Watch that there are spaces around the brackets during the test (after the while statement). These spaces are necessary.

This loop outputs:

```
i is currently 0
i is currently 1
i is currently 2
i is currently 3
```

```
i is currently 4
```

Section 10.8: For Loop with C-style syntax

The basic format of C-style **for** loop is:

```
for (( variable assignment; condition; iteration process ))
```

Notes:

- The assignment of the variable inside C-style **for** loop can contain spaces unlike the usual assignment
- Variables inside C-style **for** loop aren't preceded with $.

Example:

```
for (( i = 0; i < 10; i++ ))
do
    echo "The iteration number is $i"
done
```

Also we can process multiple variables inside C-style **for** loop:

```
for (( i = 0, j = 0; i < 10; i++, j = i * i ))
do
    echo "The square of $i is equal to $j"
done
```

Section 10.9: Until Loop

Until loop executes until condition is true

```
i=5
until [[ i -eq 10 ]]; do #Checks if i=10
    echo "i=$i" #Print the value of i
    i=$((i+1)) #Increment i by 1
done
```

Output:

```
i=5
i=6
i=7
i=8
i=9
```

When `i` reaches 10 the condition in until loop becomes true and the loop ends.

Section 10.10: Switch statement with case

With the **case** statement you can match values against one variable.

The argument passed to **case** is expanded and try to match against each patterns.

If a match is found, the commands upto `;;` are executed.

```
case "$BASH_VERSION" in
 [34]*)
    echo {1..4}
    ;;
  *)
    seq -s" " 1 4
esac
```

Pattern are not regular expressions but shell pattern matching (aka globs).

Section 10.11: For Loop without a list-of-words parameter

```
for arg; do
    echo arg=$arg
done
```

A `for` loop without a list of words parameter will iterate over the positional parameters instead. In other words, the above example is equivalent to this code:

```
for arg in "$@"; do
    echo arg=$arg
done
```

In other words, if you catch yourself writing `for i in "$@"; do ...; done`, just drop the `in` part, and write simply `for i; do ...; done`.

Chapter 11: true, false and : commands

Section 11.1: Infinite Loop

```
while true; do
    echo ok
done
```

or

```
while :; do
   echo ok
done
```

or

```
until false; do
    echo ok
done
```

Section 11.2: Function Return

```
function positive() {
    return 0
}

function negative() {
    return 1
}
```

Section 11.3: Code that will always/never be executed

```
if true; then
    echo Always executed
fi
if false; then
    echo Never executed
fi
```

Chapter 12: Arrays

Section 12.1: Array Assignments

List Assignment

If you are familiar with Perl, C, or Java, you might think that Bash would use commas to separate array elements, however this is not the case; instead, Bash uses spaces:

```
# Array in Perl
my @array = (1, 2, 3, 4);
```

```
# Array in Bash
array=(1 2 3 4)
```

Create an array with new elements:

```
array=('first element' 'second element' 'third element')
```

Subscript Assignment

Create an array with explicit element indices:

```
array=([3]='fourth element' [4]='fifth element')
```

Assignment by index

```
array[0]='first element'
array[1]='second element'
```

Assignment by name (associative array)

Version ≥ 4.0

```
declare -A array
array[first]='First element'
array[second]='Second element'
```

Dynamic Assignment

Create an array from the output of other command, for example use **seq** to get a range from 1 to 10:

```
array=(`seq 1 10`)
```

Assignment from script's input arguments:

```
array=("$@")
```

Assignment within loops:

```
while read -r; do
    #array+=("$REPLY")     # Array append
    array[$i]="$REPLY"     # Assignment by index
    let i++                # Increment index
done < <(seq 1 10)  # command substitution
```

```
echo ${array[@]}    # output: 1 2 3 4 5 6 7 8 9 10
```

where $REPLY is always the current input

Section 12.2: Accessing Array Elements

Print element at index 0

```
echo "${array[0]}"
```

Version < 4.3

Print last element using substring expansion syntax

```
echo "${arr[@]: -1 }"
```

Version ≥ 4.3

Print last element using subscript syntax

```
echo "${array[-1]}"
```

Print all elements, each quoted separately

```
echo "${array[@]}"
```

Print all elements as a single quoted string

```
echo "${array[*]}"
```

Print all elements from index 1, each quoted separately

```
echo "${array[@]:1}"
```

Print 3 elements from index 1, each quoted separately

```
echo "${array[@]:1:3}"
```

String Operations

If referring to a single element, string operations are permitted:

```
array=(zero one two)
echo "${array[0]:0:3}" # gives out zer (chars at position 0, 1 and 2 in the string zero)
echo "${array[0]:1:3}" # gives out ero (chars at position 1, 2 and 3 in the string zero)
```

so `${array[$i]:N:M}` gives out a string from the `N`th position (starting from 0) in the string `${array[$i]}` with `M` following chars.

Section 12.3: Array Modification

Change Index

Initialize or update a particular element in the array

```
array[10]="elevenths element"    # because it's starting with 0
```

Version ≥ 3.1

Append

Modify array, adding elements to the end if no subscript is specified.

```
array+=('fourth element' 'fifth element')
```

Replace the entire array with a new parameter list.

```
array=("${array[@]}" "fourth element" "fifth element")
```

Add an element at the beginning:

```
array=("new element" "${array[@]}")
```

Insert

Insert an element at a given index:

```
arr=(a b c d)
# insert an element at index 2
i=2
arr=("${arr[@]:0:$i}" 'new' "${arr[@]:$i}")
echo "${arr[2]}" #output: new
```

Delete

Delete array indexes using the `unset` builtin:

```
arr=(a b c)
echo "${arr[@]}"   # outputs: a b c
echo "${!arr[@]}"  # outputs: 0 1 2
unset -v 'arr[1]'
echo "${arr[@]}"   # outputs: a c
echo "${!arr[@]}"  # outputs: 0 2
```

Merge

```
array3=("${array1[@]}" "${array2[@]}")
```

This works for sparse arrays as well.

Re-indexing an array

This can be useful if elements have been removed from an array, or if you're unsure whether there are gaps in the array. To recreate the indices without gaps:

```
array=("${array[@]}")
```

Section 12.4: Array Iteration

Array iteration comes in two flavors, foreach and the classic for-loop:

```
a=(1 2 3 4)
# foreach loop
```

```
for y in "${a[@]}"; do
    # act on $y
    echo "$y"
done
# classic for-loop
for ((idx=0; idx < ${#a[@]}; ++idx)); do
    # act on ${a[$idx]}
    echo "${a[$idx]}"
done
```

You can also iterate over the output of a command:

```
a=($(tr ',' ' ' <<<"a,b,c,d")) # tr can transform one character to another
for y in "${a[@]}"; do
    echo "$y"
done
```

Section 12.5: Array Length

`${#array[@]}` gives the length of the array `${array[@]}`:

```
array=('first element' 'second element' 'third element')
echo "${#array[@]}" # gives out a length of 3
```

This works also with Strings in single elements:

```
echo "${#array[0]}"    # gives out the lenght of the string at element 0: 13
```

Section 12.6: Associative Arrays

Version ≥ 4.0

Declare an associative array

```
declare -A aa
```

Declaring an associative array before initialization or use is mandatory.

Initialize elements

You can initialize elements one at a time as follows:

```
aa[hello]=world
aa[ab]=cd
aa["key with space"]="hello world"
```

You can also initialize an entire associative array in a single statement:

```
aa=([hello]=world [ab]=cd ["key with space"]="hello world")
```

Access an associative array element

```
echo ${aa[hello]}
# Out: world
```

Listing associative array keys

```
echo "${!aa[@]}"
#Out: hello ab key with space
```

Listing associative array values

```
echo "${aa[@]}"
#Out: world cd hello world
```

Iterate over associative array keys and values

```
for key in "${!aa[@]}"; do
    echo "Key:   ${key}"
    echo "Value: ${array[$key]}"
done

# Out:
# Key:   hello
# Value: world
# Key:   ab
# Value: cd
# Key:   key with space
# Value: hello world
```

Count associative array elements

```
echo "${#aa[@]}"
# Out: 3
```

Section 12.7: Looping through an array

Our example array:

```
arr=(a b c d e f)
```

Using a `for..in` loop:

```
for i in "${arr[@]}"; do
    echo "$i"
done
```

Version ≥ 2.04

Using C-style **`for`** loop:

```
for ((i=0;i<${#arr[@]};i++)); do
    echo "${arr[$i]}"
done
```

Using **`while`** loop:

```
i=0
while [ $i -lt ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

Using `while` loop with numerical conditional:

```
i=0
while (( $i < ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Using an `until` loop:

```
i=0
until [ $i -ge ${#arr[@]} ]; do
    echo "${arr[$i]}"
    i=$((i + 1))
done
```

Version ≥ 2.04

Using an `until` loop with numerical conditional:

```
i=0
until (( $i >= ${#arr[@]} )); do
    echo "${arr[$i]}"
    ((i++))
done
```

Section 12.8: Destroy, Delete, or Unset an Array

To destroy, delete, or unset an array:

```
unset array
```

To destroy, delete, or unset a single array element:

```
unset array[10]
```

Section 12.9: Array from string

```
stringVar="Apple Orange Banana Mango"
arrayVar=(${stringVar// / })
```

Each space in the string denotes a new item in the resulting array.

```
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[3]} # will print Mango
```

Similarly, other characters can be used for the delimiter.

```
stringVar="Apple+Orange+Banana+Mango"
arrayVar=(${stringVar//+/ })
echo ${arrayVar[0]} # will print Apple
echo ${arrayVar[2]} # will print Banana
```

Section 12.10: List of initialized indexes

Get the list of inialized indexes in an array

```
$ arr[2]='second'
$ arr[10]='tenth'
$ arr[25]='twenty five'
$ echo ${!arr[@]}
2 10 25
```

Section 12.11: Reading an entire file into an array

Reading in a single step:

```
IFS=$'\n' read -r -a arr < file
```

Reading in a loop:

```
arr=()
while IFS= read -r line; do
  arr+=("$line")
done
```

Version ≥ 4.0

Using `mapfile` or `readarray` (which are synonymous):

```
mapfile -t arr < file
readarray -t arr < file
```

Section 12.12: Array insert function

This function will insert an element into an array at a given index:

```
insert(){
    h='
################## insert ########################
# Usage:
#   insert arr_name index element
#
#   Parameters:
#       arr_name    : Name of the array variable
#       index       : Index to insert at
#       element     : Element to insert
##################################################
    '
    [[ $1 = -h ]] && { echo "$h" >/dev/stderr; return 1; }
    declare -n __arr__=$1   # reference to the array variable
    i=$2                    # index to insert at
    el="$3"                 # element to insert
    # handle errors
    [[ ! "$i" =~ ^[0-9]+$ ]] && { echo "E: insert: index must be a valid integer" >/dev/stderr;
return 1; }
    (( $1 < 0 )) && { echo "E: insert: index can not be negative" >/dev/stderr; return 1; }
    # Now insert $el at $i
    __arr__=("${__arr__[@]:0:$i}" "$el" "${__arr__[@]:$i}")
}
```

Usage:

```
insert array_variable_name index element
```

Example:

```
arr=(a b c d)
echo "${arr[2]}" # output: c
# Now call the insert function and pass the array variable name,
# index to insert at
# and the element to insert
insert arr 2 'New Element'
# 'New Element' was inserted at index 2 in arr, now print them
echo "${arr[2]}" # output: New Element
echo "${arr[3]}" # output: c
```

Chapter 13: Associative arrays

Section 13.1: Examining assoc arrays

All needed usage shown with this snippet:

```
#!/usr/bin/env bash

declare -A assoc_array=([key_string]=value                                \
                        [one]="something"                                 \
                        [two]="another thing"                             \
                        [ three ]='mind the blanks!'                      \
                        [ " four" ]='count the blanks of this key later!' \
                                                 [IMPORTANT]='SPACES DO ADD UP!!!'
 \
                        [1]='there are no integers!'                      \
                        [info]="to avoid history expansion "              \
                        [info2]="quote exclamation mark with single quotes"  \
                       )
echo # just a blank line
echo now here are the values of assoc_array:
echo ${assoc_array[@]}
echo not that useful,
echo # just a blank line
echo this is better:

declare -p assoc_array    # -p == print

echo have a close look at the spaces above\!\!\!
echo # just a blank line

echo accessing the keys
echo the keys in assoc_array are ${!assoc_array[*]}
echo mind the use of indirection operator \!
echo # just a blank line


echo now we loop over the assoc_array line by line
echo note the \! indirection operator which works differently,
echo if used with assoc_array.
echo # just a blank line

for key in "${!assoc_array[@]}"; do # accessing keys using ! indirection!!!!
   printf "key: \"%s\"\nvalue: \"%s\"\n\n" "$key" "${assoc_array[$key]}"
done


echo have a close look at the spaces in entries with keys two, three and four above\!\!\!
echo # just a blank line
echo # just another blank line

echo there is a difference using integers as keys\!\!\!
i=1
echo declaring an integer var i=1
echo # just a blank line
echo Within an integer_array bash recognizes artithmetic context.
echo Within an assoc_array bash DOES NOT recognize artithmetic context.
echo # just a blank line
echo this works: \${assoc_array[\$i]}:  ${assoc_array[$i]}
echo this NOT!!: \${assoc_array[i]}:  ${assoc_array[i]}
```

```bash
echo # just a blank line
echo # just a blank line
echo an \${assoc_array[i]} has a string context within braces in contrast to an integer_array
declare -i integer_array=( one two three )
echo "doing a: declare -i integer_array=( one two three )"
echo # just a blank line

echo both forms do work: \${integer_array[i]} : ${integer_array[i]}
echo and this too:  \${integer_array[\$i]} : ${integer_array[$i]}
```

Chapter 14: Functions

Section 14.1: Functions with arguments

In `helloJohn.sh`:

```
#!/bin/bash

greet() {
  local name="$1"
  echo "Hello, $name"
}

greet "John Doe"
```

```
# running above script
$ bash helloJohn.sh
Hello, John Doe
```

1. If you don't modify the argument in any way, there is no need to copy it to a **`local`** variable - simply **`echo`** `"Hello, $1"`.

2. You can use `$1`, `$2`, `$3` and so on to access the arguments inside the function.

> **Note:** for arguments more than 9 `$10` won't work (bash will read it as **$1**0), you need to do `${10}`, `${11}` and so on.

3. `$@` refers to all arguments of a function:

```
#!/bin/bash
foo() {
  echo "$@"
}

foo 1 2 3 # output => 1 2 3
```

> **Note:** You should practically always use double quotes around `"$@"`, like here.

Omitting the quotes will cause the shell to expand wildcards (even when the user specifically quoted them in order to avoid that) and generally introduce unwelcome behavior and potentially even security problems.

```
foo "string with spaces;" '$HOME' "*"
# output => string with spaces; $HOME *
```

4. for default arguments use `${1:-default_val}`. Eg:

```
#!/bin/bash
foo() {
  local val=${1:-25}
  echo "$val"
}
```

```
foo     # output => 25
foo 30  # output => 30
```

5. to require an argument use `${var:?error message}`

```
foo() {
  local val=${1:?Must provide an argument}
  echo "$val"
}
```

Section 14.2: Simple Function

In `helloWorld.sh`

```
#!/bin/bash

# Define a function greet
greet ()
{
    echo "Hello World!"
}

# Call the function greet
greet
```

In running the script, we see our message

```
$ bash helloWorld.sh
Hello World!
```

Note that sourcing a file with functions makes them available in your current bash session.

```
$ source helloWorld.sh   # or, more portably, ". helloWorld.sh"
$ greet
Hello World!
```

You can **`export`** a function in some shells, so that it is exposed to child processes.

```
bash -c 'greet'  # fails
export -f greet  # export function; note -f
bash -c 'greet'  # success
```

Section 14.3: Handling flags and optional parameters

The *getopts* builtin can be used inside functions to write functions that accommodate flags and optional parameters. This presents no special difficulty but one has to handle appropriately the values touched by *getopts*. As an example, we define a *failwith* function that writes a message on *stderr* and exits with code 1 or an arbitrary code supplied as parameter to the `-x` option:

```
# failwith [-x STATUS] PRINTF-LIKE-ARGV
#  Fail with the given diagnostic message
#
# The -x flag can be used to convey a custom exit status, instead of
# the value 1.  A newline is automatically added to the output.

failwith()
```

```
{
    local OPTIND OPTION OPTARG status

    status=1
    OPTIND=1

    while getopts 'x:' OPTION; do
        case ${OPTION} in
            x)    status="${OPTARG}";;
            *)    1>&2 printf 'failwith: %s: Unsupported option.\n' "${OPTION}";;
        esac
    done

    shift $(( OPTIND - 1 ))
    {
        printf 'Failure: '
        printf "$@"
        printf '\n'
    } 1>&2
    exit "${status}"
}
```

This function can be used as follows:

```
failwith '%s: File not found.' "${filename}"
failwith -x 70 'General internal error.'
```

and so on.

Note that as for *printf*, variables should not be used as first argument. If the message to print consists of the content of a variable, one should use the `%s` specifier to print it, like in

```
failwith '%s' "${message}"
```

Section 14.4: Print the function definition

```
getfunc() {
    declare -f "$@"
}

function func(){
    echo "I am a sample function"
}

funcd="$(getfunc func)"
getfunc func # or echo "$funcd"
```

Output:

```
func ()
{
    echo "I am a sample function"
}
```

Section 14.5: A function that accepts named parameters

```
foo() {
  while [[ "$#" -gt 0 ]]
```

```
  do
    case $1 in
      -f|--follow)
        local FOLLOW="following"
        ;;
      -t|--tail)
        local TAIL="tail=$2"
        ;;
    esac
    shift
  done

  echo "FOLLOW: $FOLLOW"
  echo "TAIL: $TAIL"
}
```

Example usage:

```
foo -f
foo -t 10
foo -f --tail 10
foo --follow --tail 10
```

Section 14.6: Return value from a function

The `return` statement in Bash doesn't return a value like C-functions, instead it exits the function with a return status. You can think of it as the exit status of that function.

If you want to return a value from the function then send the value to `stdout` like this:

```
fun() {
    local var="Sample value to be returned"
    echo "$var"
    #printf "%s\n" "$var"
}
```

Now, if you do:

```
var="$(fun)"
```

the output of `fun` will be stored in `$var`.

Section 14.7: The exit code of a function is the exit code of its last command

Consider this example function to check if a host is up:

```
is_alive() {
    ping -c1 "$1" &> /dev/null
}
```

This function sends a single ping to the host specified by the first function parameter. The output and error output of `ping` are both redirected to `/dev/null`, so the function will never output anything. But the `ping` command will have exit code 0 on success, and non-zero on failure. As this is the last (and in this example, the only) command of the function, the exit code of `ping` will be reused for the exit code of the function itself.

This fact is very useful in conditional statements.

For example, if host `graucho` is up, then connect to it with **ssh**:

```
if is_alive graucho; then
    ssh graucho
fi
```

Another example: repeatedly check until host `graucho` is up, and then connect to it with **ssh**:

```
while ! is_alive graucho; do
    sleep 5
done
ssh graucho
```

Chapter 15: Bash Parameter Expansion

The $ character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

Read more in the Bash User Manual.

Section 15.1: Modifying the case of alphabetic characters

Version ≥ 4.0

To uppercase

```
$ v="hello"
# Just the first character
$ printf '%s\n' "${v^}"
Hello
# All characters
$ printf '%s\n' "${v^^}"
HELLO
# Alternative
$ v="hello world"
$ declare -u string="$v"
$ echo "$string"
HELLO WORLD
```

To lowercase

```
$ v="BYE"
# Just the first character
$ printf '%s\n' "${v,}"
bYE
# All characters
$ printf '%s\n' "${v,,}"
bye
# Alternative
$ v="HELLO WORLD"
$ declare -l string="$v"
$ echo "$string"
hello world
```

Toggle Case

```
$ v="Hello World"
# All chars
$ echo "${v~~}"
hELLO wORLD
$ echo "${v~}"
# Just the first char
hello World
```

Section 15.2: Length of parameter

```
# Length of a string
$ var='12345'
$ echo "${#var}"
```

```
5
```

Note that it's the length in number of *characters* which is not necessarily the same as the number of *bytes* (like in UTF-8 where most characters are encoded in more than one byte), nor the number of *glyphs/graphemes* (some of which are combinations of characters), nor is it necessarily the same as the display width.

```
# Number of array elements
$ myarr=(1 2 3)
$ echo "${#myarr[@]}"
3

# Works for positional parameters as well
$ set -- 1 2 3 4
$ echo "${#@}"
4

# But more commonly (and portably to other shells), one would use
$ echo "$#"
4
```

Section 15.3: Replace pattern in string

First match:

```
$ a='I am a string'
$ echo "${a/a/A}"
I Am a string
```

All matches:

```
$ echo "${a//a/A}"
I Am A string
```

Match at the beginning:

```
$ echo "${a/#I/y}"
y am a string
```

Match at the end:

```
$ echo "${a/%g/N}"
I am a strinN
```

Replace a pattern with nothing:

```
$ echo "${a/g/}"
I am a strin
```

Add prefix to array items:

```
$ A=(hello world)
$ echo "${A[@]/#/R}"
Rhello Rworld
```

Section 15.4: Substrings and subarrays

```
var='0123456789abcdef'

# Define a zero-based offset
$ printf '%s\n' "${var:3}"
3456789abcdef

# Offset and length of substring
$ printf '%s\n' "${var:3:4}"
3456
```

Version ≥ 4.2

```
# Negative length counts from the end of the string
$ printf '%s\n' "${var:3:-5}"
3456789a

# Negative offset counts from the end
# Needs a space to avoid confusion with ${var:-6}
$ printf '%s\n' "${var: -6}"
abcdef

# Alternative: parentheses
$ printf '%s\n' "${var:(-6)}"
abcdef

# Negative offset and negative length
$ printf '%s\n' "${var: -6:-5}"
a
```

The same expansions apply if the parameter is a **positional parameter** or the **element of a subscripted array**:

```
# Set positional parameter $1
set -- 0123456789abcdef

# Define offset
$ printf '%s\n' "${1:5}"
56789abcdef

# Assign to array element
myarr[0]='0123456789abcdef'

# Define offset and length
$ printf '%s\n' "${myarr[0]:7:3}"
789
```

Analogous expansions apply to **positional parameters**, where offsets are one-based:

```
# Set positional parameters $1, $2, ...
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f

# Define an offset (beware $0 (not a positional parameter)
# is being considered here as well)
$ printf '%s\n' "${@:10}"
0
a
b
c
d
e
f
```

```
# Define an offset and a length
$ printf '%s\n' "${@:10:3}"
0
a
b

# No negative lengths allowed for positional parameters
$ printf '%s\n' "${@:10:-2}"
bash: -2: substring expression < 0

# Negative offset counts from the end
# Needs a space to avoid confusion with ${@:-10:2}
$ printf '%s\n' "${@: -10:2}"
7
8

# ${@:0} is $0 which is not otherwise a positional parameters or part
# of $@
$ printf '%s\n' "${@:0:2}"
/usr/bin/bash
1
```

Substring expansion can be used with **indexed arrays**:

```
# Create array (zero-based indices)
$ myarr=(0 1 2 3 4 5 6 7 8 9 a b c d e f)

# Elements with index 5 and higher
$ printf '%s\n' "${myarr[@]:12}"
c
d
e
f

# 3 elements, starting with index 5
$ printf '%s\n' "${myarr[@]:5:3}"
5
6
7

# The last element of the array
$ printf '%s\n' "${myarr[@]: -1}"
f
```

Section 15.5: Delete a pattern from the beginning of a string

Shortest match:

```
$ a='I am a string'
$ echo "${a#*a}"
m a string
```

Longest match:

```
$ echo "${a##*a}"
 string
```

Section 15.6: Parameter indirection

Bash indirection permits to get the value of a variable whose name is contained in another variable. Variables example:

```
$ red="the color red"
$ green="the color green"

$ color=red
$ echo "${!color}"
the color red
$ color=green
$ echo "${!color}"
the color green
```

Some more examples that demonstrate the indirect expansion usage:

```
 $ foo=10
 $ x=foo
 $ echo ${x}      #Classic variable print
 foo

 $ foo=10
 $ x=foo
 $ echo ${!x}     #Indirect expansion
 10
```

One more example:

```
$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${i}";done; }; argtester -ab -cd -ef
1   #i expanded to 1
2   #i expanded to 2
3   #i expanded to 3

$ argtester () { for (( i=1; i<="$#"; i++ )); do echo "${!i}";done; }; argtester -ab -cd -ef
-ab     # i=1 --> expanded to $1 ---> expanded to first argument sent to function
-cd     # i=2 --> expanded to $2 ---> expanded to second argument sent to function
-ef     # i=3 --> expanded to $3 ---> expanded to third argument sent to function
```

Section 15.7: Parameter expansion and filenames

You can use Bash Parameter Expansion to emulate common filename-processing operations like `basename` and `dirname`.

We will use this as our example path:

```
FILENAME="/tmp/example/myfile.txt"
```

To emulate `dirname` and return the directory name of a file path:

```
echo "${FILENAME%/*}"
#Out: /tmp/example
```

To emulate `basename $FILENAME` and return the filename of a file path:

```
echo "${FILENAME##*/}"
```

```
#Out: myfile.txt
```

To emulate `basename $FILENAME .txt` and return the filename without the `.txt.` extension:

```
BASENAME="${FILENAME##*/}"
echo "${BASENAME%%.txt}"
#Out: myfile
```

Section 15.8: Default value substitution

> `${parameter:-word}`
>
> If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

```
$ unset var
$ echo "${var:-XX}"     # Parameter is unset -> expansion XX occurs
XX
$ var=""                # Parameter is null -> expansion XX occurs
$ echo "${var:-XX}"
XX
$ var=23                # Parameter is not null -> original expansion occurs
$ echo "${var:-XX}"
23
```

> `${parameter:=word}`
>
> If parameter is unset or null, the expansion of word is assigned to parameter. The value of parameter is then substituted. Positional parameters and special parameters may not be assigned to in this way.

```
$ unset var
$ echo "${var:=XX}"     # Parameter is unset -> word is assigned to XX
XX
$ echo "$var"
XX
$ var=""                # Parameter is null -> word is assigned to XX
$ echo "${var:=XX}"
XX
$ echo "$var"
XX
$ var=23                # Parameter is not null -> no assignment occurs
$ echo "${var:=XX}"
23
$ echo "$var"
23
```

Section 15.9: Delete a pattern from the end of a string

Shortest match:

```
$ a='I am a string'
$ echo "${a%a*}"
I am
```

Longest match:

```
$ echo "${a%%a*}"
I
```

Section 15.10: Munging during expansion

Variables don't necessarily have to expand to their values - substrings can be extracted during expansion, which can be useful for extracting file extensions or parts of paths. Globbing characters keep their usual meanings, so `.*` refers to a literal dot, followed by any sequence of characters; it's not a regular expression.

```
$ v=foo-bar-baz
$ echo ${v%%-*}
foo
$ echo ${v%-*}
foo-bar
$ echo ${v##*-}
baz
$ echo ${v#*-}
bar-baz
```

It's also possible to expand a variable using a default value - say I want to invoke the user's editor, but if they've not set one I'd like to give them `vim`.

```
$ EDITOR=nano
$ ${EDITOR:-vim} /tmp/some_file
# opens nano
$ unset EDITOR
$ $ ${EDITOR:-vim} /tmp/some_file
# opens vim
```

There are two different ways of performing this expansion, which differ in whether the relevant variable is empty or unset. Using `:-` will use the default if the variable is either unset or empty, whilst `-` only uses the default if the variable is unset, but will use the variable if it is set to the empty string:

```
$ a="set"
$ b=""
$ unset c
$ echo ${a:-default_a} ${b:-default_b} ${c:-default_c}
set default_b default_c
$ echo ${a-default_a} ${b-default_b} ${c-default_c}
set default_c
```

Similar to defaults, alternatives can be given; where a default is used if a particular variable isn't available, an alternative is used if the variable is available.

```
$ a="set"
$ b=""
$ echo ${a:+alternative_a} ${b:+alternative_b}
alternative_a
```

Noting that these expansions can be nested, using alternatives becomes particularly useful when supplying arguments to command line flags;

```
$ output_file=/tmp/foo
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget -o /tmp/foo www.stackexchange.com
$ unset output_file
```

```
$ wget ${output_file:+"-o ${output_file}"} www.stackexchange.com
# expands to wget  www.stackexchange.com
```

Section 15.11: Error if variable is empty or unset

The semantics for this are similar to that of default value substitution, but instead of substituting a default value, it errors out with the provided error message. The forms are `${VARNAME?ERRMSG}` and `${VARNAME:?ERRMSG}`. The form with `:` will error our if the variable is **unset** *or* **empty**, whereas the form without will only error out if the variable is *unset*. If an error is thrown, the `ERRMSG` is output and the exit code is set to `1`.

```
#!/bin/bash
FOO=
# ./script.sh: line 4: FOO: EMPTY
echo "FOO is ${FOO:?EMPTY}"
# FOO is
echo "FOO is ${FOO?UNSET}"
# ./script.sh: line 8: BAR: EMPTY
echo "BAR is ${BAR:?EMPTY}"
# ./script.sh: line 10: BAR: UNSET
echo "BAR is ${BAR?UNSET}"
```

The run the full example above each of the erroring echo statements needs to be commented out to proceed.

Chapter 16: Copying (cp)

Option	Description
-a,-archive	Combines the d, p and r options
-b, -backup	Before removal, makes a backup
-d, --no-deference	Preserves links
-f, --force	Remove existing destinations without prompting user
-i, --interactive	Show prompt before overwriting
-l, --link	Instead of copying, link files instead
-p, --preserve	Preserve file attributes when possible
-R, --recursive	Recursively copy directories

Section 16.1: Copy a single file

Copy foo.txt from /path/to/source/ to /path/to/target/folder/

```
cp /path/to/source/foo.txt /path/to/target/folder/
```

Copy foo.txt from /path/to/source/ to /path/to/target/folder/ into a file called bar.txt

```
cp /path/to/source/foo.txt /path/to/target/folder/bar.txt
```

Section 16.2: Copy folders

copy folder foo into folder bar

```
cp -r /path/to/foo /path/to/bar
```

if folder bar exists before issuing the command, then foo and its content will be copied into the folder bar. However, if bar does not exist before issuing the command, then the folder bar will be created and the content of foo will be placed into bar

Chapter 17: Find

find is a command to recursively search a directory for files(or directories) that match a criteria, and then perform some action on the selected files.

find search_path selection_criteria action

Section 17.1: Searching for a file by name or extension

To find files/directories with a specific name, relative to **pwd**:

```
$ find . -name "myFile.txt"
./myFile.txt
```

To find files/directories with a specific extension, use a wildcard:

```
$ find . -name "*.txt"
./myFile.txt
./myFile2.txt
```

To find files/directories matching one of many extensions, use the `or` flag:

```
$ find . -name "*.txt" -o -name "*.sh"
```

To find files/directories which name begin with abc and end with one alpha character following a one digit:

```
$ find . -name "abc[a-z][0-9]"
```

To find all files/directories located in a specific directory

```
$ find /opt
```

To search for files only (not directories), use `-type f`:

```
find /opt -type f
```

To search for directories only (not regular files), use `-type d`:

```
find /opt -type d
```

Section 17.2: Executing commands against a found file

Sometimes we will need to run commands against a lot of files. This can be done using **xargs**.

```
find . -type d -print | xargs -r chmod 770
```

The above command will recursively find all directories (`-type d`) relative to `.` (which is your current working directory), and execute **chmod** `770` on them. The `-r` option specifies to **xargs** to not run **chmod** if **find** did not find any files.

If your files names or directories have a space character in them, this command may choke; a solution is to use the following

```
find . -type d -print0 | xargs -r -0 chmod 770
```

In the above example, the `-print0` and `-0` flags specify that the file names will be separated using a `null` byte, and allows the use of special characters, like spaces, in the file names. This is a GNU extension, and may not work in other versions of `find` and `xargs`.

The preferred way to do this is to skip the `xargs` command and let `find` call the subprocess itself:

```
find . -type d -exec chmod 770 {} \;
```

Here, the `{}` is a placeholder indicating that you want to use the file name at that point. `find` will execute `chmod` on each file individually.

You can alternatively pass all file names to a *single* call of `chmod`, by using

```
find . -type d -exec chmod 770 {} +
```

This is also the behaviour of the above `xargs` snippets. (To call on each file individually, you can use `xargs -n1`).

A third option is to let bash loop over the list of filenames `find` outputs:

```
find . -type d | while read -r d; do chmod 770 "$d"; done
```

This is syntactically the most clunky, but convenient when you want to run multiple commands on each found file. However, this is **unsafe** in the face of file names with odd names.

```
find . -type f | while read -r d; do mv "$d" "${d// /_}"; done
```

which will replace all spaces in file names with underscores.(This example also won't work if there are spaces in leading *directory* names.)

The problem with the above is that `while read -r` expects one entry per line, but file names can contain newlines (and also, `read -r` will lose any trailing whitespace). You can fix this by turning things around:

```
find . -type d -exec bash -c 'for f; do mv "$f" "${f// /_}"; done' _ {} +
```

This way, the `-exec` receives the file names in a form which is completely correct and portable; the `bash -c` receives them as a number of arguments, which will be found in `$@`, correctly quoted etc. (The script will need to handle these names correctly, of course; every variable which contains a file name needs to be in double quotes.)

The mysterious `_` is necessary because the first argument to `bash -c 'script'` is used to populate `$0`.

Section 17.3: Finding file by access / modification time

On an `ext` filesystem, each file has a stored Access, Modification, and (Status) Change time associated with it - to view this information you can use `stat myFile.txt`; using flags within *find*, we can search for files that were modified within a certain time range.

To find files that *have* been modified within the last 2 hours:

```
$ find . -mmin -120
```

To find files that *have not* been modified within the last 2 hours:

```
$ find . -mmin +120
```

The above example are searching only on the *modified* time - to search on **a**ccess times, or **c**hanged times, use `a`, or `c` accordingly.

```
$ find . -amin -120
$ find . -cmin +120
```

General format:

`-mmin n` : File was modified *n* minutes ago
`-mmin -n` : File was modified less than *n* minutes ago
`-mmin +n` : File was modified more than *n* minutes ago

Find files that *have* been modified within the last 2 days:

```
find . -mtime -2
```

Find files that *have not* been modified within the last 2 days

```
find . -mtime +2
```

Use `-atime` and `-ctime` for access time and status change time respectively.

General format:

`-mtime n` : File was modified *nx24* hours ago
`-mtime -n` : File was modified less than *nx24* hours ago
`-mtime +n` : File was modified more than *nx24* hours ago

Find files modified in a ***range of dates***, from 2007-06-07 to 2007-06-08:

```
find . -type f -newermt 2007-06-07 ! -newermt 2007-06-08
```

Find files accessed in a ***range of timestamps*** (using files as timestamp), from 1 hour ago to 10 minutes ago:

```
touch -t $(date -d '1 HOUR AGO' +%Y%m%d%H%M.%S) start_date
touch -t $(date -d '10 MINUTE AGO' +%Y%m%d%H%M.%S) end_date
timeout 10 find "$LOCAL_FOLDER" -newerat "start_date" ! -newerat "end_date" -print
```

General format:

`-newerXY reference` : Compares the timestamp of the current file with reference. `XY` could have one of the following values: `at` (access time), `mt` (modification time), `ct` (change time) and more. `reference` is the *name of a file* whe want to compare the timestamp specified (access, modification, change) or a *string* describing an absolute time.

Section 17.4: Finding files according to size

Find files larger than 15MB:

```
find -type f -size +15M
```

Find files less than 12KB:

```
find -type f -size -12k
```

Find files exactly of 12KB size:

```
find -type f -size 12k
```

Or

```
find -type f -size 12288c
```

Or

```
find -type f -size 24b
```

Or

```
find -type f -size 24
```

General format:

```
find [options] -size n[cwbkMG]
```

> Find files of n-block size, where +n means more than n-block, -n means less than n-block and n (without any sign) means exactly n-block

Block size:

1. `c`: bytes
2. `w`: 2 bytes
3. `b`: 512 bytes (default)
4. `k`: 1 KB
5. `M`: 1 MB
6. `G`: 1 GB

Section 17.5: Filter the path

The `-path` parameter allows to specify a pattern to match the path of the result. The pattern can match also the name itself.

To find only files containing `log` anywhere in their path (folder or name):

```
find . -type f -path '*log*'
```

To find only files within a folder called `log` (on any level):

```
find . -type f -path '*/log/*'
```

To find only files within a folder called `log` or `data`:

```
find . -type f -path '*/log/*' -o -path '*/data/*'
```

To find all files **except** the ones contained in a folder called `bin`:

```
find . -type f -not -path '*/bin/*'
```

To find all file all files **except** the ones contained in a folder called `bin` or log files:

```
find . -type f -not -path '*log' -not -path '*/bin/*'
```

Section 17.6: Finding files by type

To find files, use the `-type f` flag

```
$ find . -type f
```

To find directories, use the `-type d` flag

```
$ find . -type d
```

To find block devices, use the `-type b` flag

```
$ find /dev -type b
```

To find symlinks, use the `-type l` flag

```
$ find . -type l
```

Section 17.7: Finding files by specific extension

To find all the files of a certain extension within the current path you can use the following `find` syntax. It works by making use of `bash's` built-in `glob` construct to match all the names having the `.extension`.

```
find /directory/to/search -maxdepth 1 -type f -name "*.extension"
```

To find all files of type `.txt` from the current directory alone, do

```
find . -maxdepth 1 -type f -name "*.txt"
```

Chapter 18: Using sort

Option	Meaning
-u	Make each lines of output unique

sort is a Unix command to order data in file(s) in a sequence.

Section 18.1: Sort command output

`sort` command is used to sort a list of lines.

Input from a file

```
sort file.txt
```

Input from a command

You can sort any output command. In the example a list of file following a pattern.

```
find * -name pattern | sort
```

Section 18.2: Make output unique

If each lines of the output need to be unique, add `-u` option.

To display owner of files in folder

```
ls -l | awk '{print $3}' | sort -u
```

Section 18.3: Numeric sort

Suppose we have this file:

```
test>>cat file
10.Gryffindor
4.Hogwarts
2.Harry
3.Dumbledore
1.The sorting hat
```

To sort this file numerically, use sort with -n option:

```
test>>sort -n file
```

This should sort the file as below:

```
1.The sorting hat
2.Harry
3.Dumbledore
4.Hogwarts
10.Gryffindor
```

Reversing sort order: To reverse the order of the sort use the -r option

To reverse the sort order of the above file use:

```
sort -rn file
```

This should sort the file as below:

```
10.Gryffindor
4.Hogwarts
3.Dumbledore
2.Harry
1.The sorting hat
```

Section 18.4: Sort by keys

Suppose we have this file:

```
test>>cat Hogwarts
  Harry         Malfoy        Rowena        Helga
  Gryffindor    Slytherin     Ravenclaw     Hufflepuff
  Hermione      Goyle         Lockhart      Tonks
  Ron           Snape         Olivander     Newt
  Ron           Goyle         Flitwick      Sprout
```

To sort this file using a column as key use the k option:

```
test>>sort -k 2 Hogwarts
```

This will sort the file with column 2 as the key:

```
  Ron         Goyle         Flitwick        Sprout
  Hermione    Goyle         Lockhart        Tonks
  Harry       Malfoy        Rowena          Helga
  Gryffindor Slytherin      Ravenclaw       Hufflepuff
  Ron         Snape         Olivander       Newt
```

Now if we have to sort the file with a secondary key along with the primary key use:

```
sort -k 2,2 -k 1,1 Hogwarts
```

This will first sort the file with column 2 as primary key, and then sort the file with column 1 as secondary key:

```
  Hermione        Goyle         Lockhart        Tonks
  Ron             Goyle         Flitwick        Sprout
  Harry           Malfoy        Rowena          Helga
  Gryffindor      Slytherin     Ravenclaw       Hufflepuff
  Ron             Snape         Olivander       Newt
```

If we need to sort a file with more than 1 key , then for every -k option we need to specify where the sort ends. So -k1,1 means start the sort at the first column and end sort at first column.

-t option

In the previous example the file had the default delimeter - tab. In case of sorting a file that has non-default delimeter we need the -t option to specify the delimeter. Suppose we have the file as below:

```
test>>cat file
```

```
5.|Gryffindor
4.|Hogwarts
2.|Harry
3.|Dumbledore
1.|The sorting hat
```

To sort this file as per the second column, use:

```
test>>sort -t "|" -k 2 file
```

This will sort the file as below:

```
3.|Dumbledore
5.|Gryffindor
2.|Harry
4.|Hogwarts
1.|The sorting hat
```

Chapter 19: Sourcing

Section 19.1: Sourcing a file

Sourcing a file is different from execution, in that all commands are evaluated within the context of the current bash session - this means that any variables, function, or aliases defined will persist throughout your session.

Create the file you wish to source `sourceme.sh`

```
#!/bin/bash

export A="hello_world"
alias sayHi="echo Hi"
sayHello() {
    echo Hello
}
```

From your session, source the file

```
$ source sourceme.sh
```

From hencefourth, you have all the resources of the sourced file available

```
$ echo $A
hello_world

$ sayHi
Hi

$ sayHello
Hello
```

Note that the command `.` is synonymous to `source`, such that you can simply use

```
$ . sourceme.sh
```

Section 19.2: Sourcing a virtual environment

When developing several applications on one machine, it becomes useful to separate out dependencies into virtual environments.

With the use of `virtualenv`, these environments are sourced into your shell so that when you run a command, it comes from that virtual environment.

This is most commonly installed using `pip`.

```
pip install https://github.com/pypa/virtualenv/tarball/15.0.2
```

Create a new environment

```
virtualenv --python=python3.5 my_env
```

Activate the environment

```
source my_env/bin/activate
```

Chapter 20: Here documents and here strings

Section 20.1: Execute command with here document

```
ssh -p 21 example@example.com <<EOF
  echo 'printing pwd'
  echo "\$(pwd)"
  ls -a
  find '*.txt'
EOF
```

$ is escaped because we do not want it to be expanded by the current shell i.e `$(pwd)` is to be executed on the remote shell.

Another way:

```
ssh -p 21 example@example.com <<'EOF'
  echo 'printing pwd'
  echo "$(pwd)"
  ls -a
  find '*.txt'
EOF
```

Note: The closing EOF **should** be at the beginning of the line (No whitespaces before). If indentation is required, tabs may be used if you start your heredoc with `<<-`. See the Indenting here documents and Limit Strings examples for more information.

Section 20.2: Indenting here documents

You can indent the text inside here documents with tabs, you need to use the `<<-` redirection operator instead of `<<`:

```
$ cat <<- EOF
    This is some content indented with tabs `\t`.
    You cannot indent with spaces you __have__ to use tabs.
    Bash will remove empty space before these lines.
    __Note__: Be sure to replace spaces with tabs when copying this example.
EOF

This is some content indented with tabs _\t_.
You cannot indent with spaces you __have__ to use tabs.
Bash will remove empty space before these lines.
__Note__: Be sure to replace spaces with tabs when copying this example.
```

One practical use case of this (as mentioned in `man bash`) is in shell scripts, for example:

```
if cond; then
    cat <<- EOF
    hello
    there
    EOF
fi
```

It is customary to indent the lines within code blocks as in this `if` statement, for better readability. Without the `<<-`

operator syntax, we would be forced to write the above code like this:

```
if cond; then
    cat << EOF
hello
there
EOF
fi
```

That's very unpleasant to read, and it gets much worse in a more complex realistic script.

Section 20.3: Create a file

A classic use of here documents is to create a file by typing its content:

```
cat > fruits.txt << EOF
apple
orange
lemon
EOF
```

The here-document is the lines between the `<< EOF` and `EOF`.

This here document becomes the input of the `cat` command. The `cat` command simply outputs its input, and using the output redirection operator `>` we redirect to a file `fruits.txt`.

As a result, the `fruits.txt` file will contain the lines:

```
apple
orange
lemon
```

The usual rules of output redirection apply: if `fruits.txt` did not exist before, it will be created. If it existed before, it will be truncated.

Section 20.4: Here strings

`Version ≥ 2.05b`

You can feed a command using here strings like this:

```
$ awk '{print $2}' <<< "hello world - how are you?"
world

$ awk '{print $1}' <<< "hello how are you
> she is fine"
hello
she
```

You can also feed a **while** loop with a here string:

```
$ while IFS=" " read -r word1 word2 rest
> do
> echo "$word1"
> done <<< "hello how are you - i am fine"
hello
```

Section 20.5: Run several commands with sudo

```
sudo -s <<EOF
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "\$a"
EOF
```

- `$a` needs to be escaped to prevent it to be expanded by the current shell

Or

```
sudo -s <<'EOF'
  a='var'
  echo 'Running serveral commands with sudo'
  mktemp -d
  echo "$a"
EOF
```

Section 20.6: Limit Strings

A heredoc uses the *limitstring* to determine when to stop consuming input. The terminating limitstring **must**

- Be at the start of a line.
- Be the only text on the line **Note:** If you use `<<-` the limitstring can be prefixed with tabs `\t`

Correct:

```
cat <<limitstring
line 1
line 2
limitstring
```

This will output:

> ```
> line 1
> line 2
> ```

Incorrect use:

```
cat <<limitstring
line 1
line 2
 limitstring
```

Since `limitstring` on the last line is not exactly at the start of the line, the shell will continue to wait for further input, until it sees a line that starts with `limitstring` and doesn't contain anything else. Only then it will stop waiting for input, and proceed to pass the here-document to the `cat` command.

Note that when you prefix the initial limitstring with a hyphen, any tabs at the start of the line are removed before parsing, so the data and the limit string can be indented with tabs (for ease of reading in shell scripts).

```
cat <<-limitstring
        line 1    has a tab each before the words line and has
            line 2 has two leading tabs
```

```
	limitstring
```

will produce

```
line 1	has a tab each before the words line and has
line 2 has two leading tabs
```

with the leading tabs (but not the internal tabs) removed.

Chapter 21: Quoting

Section 21.1: Double quotes for variable and command substitution

Variable substitutions should only be used inside double quotes.

```
calculation='2 * 3'
echo "$calculation"         # prints 2 * 3
echo $calculation           # prints 2, the list of files in the current directory, and 3
echo "$(($calculation))"    # prints 6
```

Outside of double quotes, `$var` takes the value of `var`, splits it into whitespace-delimited parts, and interprets each part as a glob (wildcard) pattern. Unless you want this behavior, always put `$var` inside double quotes: `"$var"`.

The same applies to command substitutions: `"$(mycommand)"` is the output of `mycommand`, `$(mycommand)` is the result of split+glob on the output.

```
echo "$var"             # good
echo "$(mycommand)"     # good
another=$var            # also works, assignment is implicitly double-quoted
make -D THING=$var      # BAD! This is not a bash assignment.
make -D THING="$var"    # good
make -D "THING=$var"    # also good
```

Command substitutions get their own quoting contexts. Writing arbitrarily nested substitutions is easy because the parser will keep track of nesting depth instead of greedily searching for the first `"` character. The StackOverflow syntax highlighter parses this wrong, however. For example:

```
echo "formatted text: $(printf "a + b = %04d" "${c}")" # “formatted text: a + b = 0000”
```

Variable arguments to a command substitution should be double-quoted inside the expansions as well:

```
echo "$(mycommand "$arg1" "$arg2")"
```

Section 21.2: Difference between double quote and single quote

Double quote	Single quote
Allows variable expansion	Prevents variable expansion
Allows history expansion if enabled	Prevents history expansion
Allows command substitution	Prevents command substitution
`*` and `@` can have special meaning	`*` and `@` are always literals
Can contain both single quote or double quote	Single quote is not allowed inside single quote
`$`, `` ` ``, `"`, `\` can be escaped with `\` to prevent their special meaning	All of them are literals

Properties that are common to both:

- Prevents globbing
- Prevents word splitting

Examples:

```
$ echo "!cat"
echo "cat file"
cat file
$ echo '!cat'
!cat
echo "\"'\""
"'"
$ a='var'
$ echo '$a'
$a
$ echo "$a"
var
```

Section 21.3: Newlines and control characters

A newline can be included in a single-quoted string or double-quoted string. Note that backslash-newline does not result in a newline, the line break is ignored.

```
newline1='
'
newline2="
"
newline3=$'\n'
empty=\

echo "Line${newline1}break"
echo "Line${newline2}break"
echo "Line${newline3}break"
echo "No line break${empty} here"
```

Inside dollar-quote strings, backslash-letter or backslash-octal can be used to insert control characters, like in many other programming languages.

```
echo $'Tab: [\t]'
echo $'Tab again: [\009]'
echo $'Form feed: [\f]'
echo $'Line\nbreak'
```

Section 21.4: Quoting literal text

All the examples in this paragraph print the line

```
!"#$&'()*;<=>?  @[\]^`{|}~
```

A backslash quotes the next character, i.e. the next character is interpreted literally. The one exception is a newline: backslash-newline expands to the empty string.

```
echo \!\"\#\$\&\'\(\)\*\;\<\=\>\?\ \ \@\[\\\]\^\`\{\|\}\~
```

All text between single quotes (forward quotes `'`, also known as apostrophe) is printed literally. Even backslash stands for itself, and it's impossible to include a single quote; instead, you can stop the literal string, include a literal single quote with a backslash, and start the literal string again. Thus the 4-character sequence `'\''` effectively allow to include a single quote in a literal string.

```
echo '!"#$&'\''()*;<=>?  @[\]^`{|}~'
```

```
#          ^^^^
```

Dollar-single-quote starts a string literal `$'...'` like many other programming languages, where backslash quotes the next character.

```
echo $'!"#$&\'()*;<=>?  @[\\]^`{|}~'
#           ^^              ^^
```

Double quotes `"` delimit semi-literal strings where only the characters `"` `\` `$` and `` ` `` retain their special meaning. These characters need a backslash before them (note that if backslash is followed by some other character, the backslash remains). Double quotes are mostly useful when including a variable or a command substitution.

```
echo "!\"#\$&'()*;<=>?  @[\\]^\`{|}~"
#      ^^                 ^^   ^^
echo "!\"#\$&'()*;<=>?  @[\]^\`{|}~"
#      ^^                ^   ^^      \[ prints \[
```

Interactively, beware that `!` triggers history expansion inside double quotes: `"!oops"` looks for an older command containing `oops`; `"\!oops"` doesn't do history expansion but keeps the backslash. This does not happen in scripts.

Chapter 22: Conditional Expressions

Section 22.1: File type tests

The `-e` conditional operator tests whether a file exists (including all file types: directories, etc.).

```
if [[ -e $filename ]]; then
  echo "$filename exists"
fi
```

There are tests for specific file types as well.

```
if [[ -f $filename ]]; then
  echo "$filename is a regular file"
elif [[ -d $filename ]]; then
  echo "$filename is a directory"
elif [[ -p $filename ]]; then
  echo "$filename is a named pipe"
elif [[ -S $filename ]]; then
  echo "$filename is a named socket"
elif [[ -b $filename ]]; then
  echo "$filename is a block device"
elif [[ -c $filename ]]; then
  echo "$filename is a character device"
fi
if [[ -L $filename ]]; then
  echo "$filename is a symbolic link (to any file type)"
fi
```

For a symbolic link, apart from `-L`, these tests apply to the target, and return false for a broken link.

```
if [[ -L $filename || -e $filename ]]; then
  echo "$filename exists (but may be a broken symbolic link)"
fi

if [[ -L $filename && ! -e $filename ]]; then
  echo "$filename is a broken symbolic link"
fi
```

Section 22.2: String comparison and matching

String comparison uses the `==` operator between *quoted* strings. The `!=` operator negates the comparison.

```
if [[ "$string1" == "$string2" ]]; then
  echo "\$string1 and \$string2 are identical"
fi
if [[ "$string1" != "$string2" ]]; then
  echo "\$string1 and \$string2 are not identical"
fi
```

If the right-hand side is not quoted then it is a wildcard pattern that `$string1` is matched against.

```
string='abc'
pattern1='a*'
pattern2='x*'
if [[ "$string" == $pattern1 ]]; then
  # the test is true
```

```
  echo "The string $string matches the pattern $pattern"
fi
if [[ "$string" != $pattern2 ]]; then
  # the test is false
  echo "The string $string does not match the pattern $pattern"
fi
```

The < and > operators compare the strings in lexicographic order (there are no less-or-equal or greater-or-equal operators for strings).

There are unary tests for the empty string.

```
if [[ -n "$string" ]]; then
  echo "$string is non-empty"
fi
if [[ -z "${string// }" ]]; then
  echo "$string is empty or contains only spaces"
fi
if [[ -z "$string" ]]; then
  echo "$string is empty"
fi
```

Above, the `-z` check may mean `$string` is unset, or it is set to an empty string. To distinguish between empty and unset, use:

```
if [[ -n "${string+x}" ]]; then
    echo "$string is set, possibly to the empty string"
fi
if [[ -n "${string-x}" ]]; then
    echo "$string is either unset or set to a non-empty string"
fi
if [[ -z "${string+x}" ]]; then
    echo "$string is unset"
fi
if [[ -z "${string-x}" ]]; then
    echo "$string is set to an empty string"
fi
```

where `x` is arbitrary. Or in table form:

```

                        +-------+-------+-----------+
$string is: | unset | empty | non-empty |
+-----------------------+-------+-------+-----------+
| [[ -z ${string} ]]    | true  | true  | false     |
| [[ -z ${string+x} ]]  | true  | false | false     |
| [[ -z ${string-x} ]]  | false | true  | false     |
| [[ -n ${string} ]]    | false | false | true      |
| [[ -n ${string+x} ]]  | false | true  | true      |
| [[ -n ${string-x} ]]  | true  | false | true      |
+-----------------------+-------+-------+-----------+
```

Alternatively, the state can be checked in a case statement:

```
case ${var+x$var} in
  (x) echo empty;;
  ("") echo unset;;
  (x*[![:blank:]]*) echo non-blank;;
  (*) echo blank
```

```
esac
```

Where `[:blank:]` is locale specific horizontal spacing characters (tab, space, etc).

Section 22.3: Test on exit status of a command

Exit status 0: success
Exit status other than 0: failure

To test on the exit status of a command:

```
if command;then
    echo 'success'
else
    echo 'failure'
fi
```

Section 22.4: One liner test

You can do things like this:

```
[[ $s = 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s == 'something' ]] && echo 'matched' || echo "didn't match"
[[ $s != 'something' ]] && echo "didn't match" || echo "matched"
[[ $s -eq 10 ]] && echo 'equal' || echo "not equal"
(( $s == 10 )) && echo 'equal' || echo 'not equal'
```

One liner test for exit status:

```
command && echo 'exited with 0' || echo 'non 0 exit'
cmd && cmd1 && echo 'previous cmds were successful' || echo 'one of them failed'
cmd || cmd1 #If cmd fails try cmd1
```

Section 22.5: File comparison

```
if [[ $file1 -ef $file2 ]]; then
  echo "$file1 and $file2 are the same file"
fi
```

"Same file" means that modifying one of the files in place affects the other. Two files can be the same even if they have different names, for example if they are hard links, or if they are symbolic links with the same target, or if one is a symbolic link pointing to the other.

If two files have the same content, but they are distinct files (so that modifying one does not affect the other), then `-ef` reports them as different. If you want to compare two files byte by byte, use the `cmp` utility.

```
if cmp -s -- "$file1" "$file2"; then
  echo "$file1 and $file2 have identical contents"
else
  echo "$file1 and $file2 differ"
fi
```

To produce a human-readable list of differences between text files, use the `diff` utility.

```
if diff -u "$file1" "$file2"; then
```

```
  echo "$file1 and $file2 have identical contents"
else
  : # the differences between the files have been listed
fi
```

Section 22.6: File access tests

```
if [[ -r $filename ]]; then
  echo "$filename is a readable file"
fi
if [[ -w $filename ]]; then
  echo "$filename is a writable file"
fi
if [[ -x $filename ]]; then
  echo "$filename is an executable file"
fi
```

These tests take permissions and ownership into account to determine whether the script (or programs launched from the script) can access the file.

Beware of race conditions (TOCTOU): just because the test succeeds now doesn't mean that it's still valid on the next line. It's usually better to try to access a file, and handle the error, rather than test first and then have to handle the error anyway in case the file has changed in the meantime.

Section 22.7: Numerical comparisons

Numerical comparisons use the `-eq` operators and friends

```
if [[ $num1 -eq $num2 ]]; then
  echo "$num1 == $num2"
fi
if [[ $num1 -le $num2 ]]; then
  echo "$num1 <= $num2"
fi
```

There are six numeric operators:

- `-eq` equal
- `-ne` not equal
- `-le` less or equal
- `-lt` less than
- `-ge` greater or equal
- `-gt` greater than

Note that the `<` and `>` operators inside `[[ … ]]` compare strings, not numbers.

```
if [[ 9 -lt 10 ]]; then
  echo "9 is before 10 in numeric order"
fi
if [[ 9 > 10 ]]; then
  echo "9 is after 10 in lexicographic order"
fi
```

The two sides must be numbers written in decimal (or in octal with a leading zero). Alternatively, use the `((…))` arithmetic expression syntax, which performs **integer** calculations in a C/Java/…-like syntax.

```bash
x=2
if ((2*x == 4)); then
  echo "2 times 2 is 4"
fi
((x += 1))
echo "2 plus 1 is $x"
```

Chapter 23: Scripting with Parameters

Section 23.1: Multiple Parameter Parsing

To parse lots of parameters, the preferred way of doing this is using a *while* loop, a *case* statement, and *shift*.

shift is used to pop the first parameter in the series, making what used to be *$2*, now be *$1*. This is useful for processing arguments one at a time.

```
#!/bin/bash

# Load the user defined parameters
while [[ $# > 0 ]]
do
        case "$1" in

                -a|--valueA)
                        valA="$2"
                        shift
                        ;;

                -b|--valueB)
                        valB="$2"
                        shift
                        ;;

                --help|*)
                        echo "Usage:"
                        echo "    --valueA \"value\""
                        echo "    --valueB \"value\""
                        echo "    --help"
                        exit 1
                        ;;
        esac
        shift
done

echo "A: $valA"
echo "B: $valB"
```

Inputs and Outputs

```
$ ./multipleParams.sh --help
Usage:
    --valueA "value"
    --valueB "value"
    --help

$ ./multipleParams.sh
A:
B:

$ ./multipleParams.sh --valueB 2
A:
B: 2

$ ./multipleParams.sh --valueB 2 --valueA "hello world"
A: hello world
```

```
B: 2
```

Section 23.2: Argument parsing using a for loop

A simple example which provides the options:

Opt	Alt. Opt	Details
`-h`	`--help`	Show help
`-v`	`--version`	Show version info
`-dr path`	`--doc-root path`	An option which takes a secondary parameter (a path)
`-i`	`--install`	A boolean option (true/false)
`-*`	--	Invalid option

```
#!/bin/bash
dr=''
install=false

skip=false
for op in "$@";do
    if $skip;then skip=false;continue;fi
    case "$op" in
        -v|--version)
            echo "$ver_info"
            shift
            exit 0
            ;;
        -h|--help)
            echo "$help"
            shift
            exit 0
            ;;
        -dr|--doc-root)
            shift
            if [[ "$1" != "" ]]; then
                dr="${1/%\//}"
                shift
                skip=true
            else
                echo "E: Arg missing for -dr option"
                exit 1
            fi
            ;;
        -i|--install)
            install=true
            shift
            ;;
        -*)
            echo "E: Invalid option: $1"
            shift
            exit 1
            ;;
    esac
done
```

Section 23.3: Wrapper script

Wrapper script is a script that wraps another script or command to provide extra functionalities or just to make something less tedious.

For example, the actual `egrep` in new GNU/Linux system is being replaced by a wrapper script named `egrep`. This is how it looks:

```
#!/bin/sh
exec grep -E "$@"
```

So, when you run `egrep` in such systems, you are actually running `grep -E` with all the arguments forwarded.

In general case, if you want to run an example script/command `exmp` with another script `mexmp` then the wrapper `mexmp` script will look like:

```
#!/bin/sh
exmp "$@" # Add other options before "$@"
# or
#full/path/to/exmp "$@"
```

Section 23.4: Accessing Parameters

When executing a Bash script, parameters passed into the script are named in accordance to their position: `$1` is the name of the first parameter, `$2` is the name of the second parameter, and so on.

A missing parameter simply evaluates to an empty string. Checking for the existence of a parameter can be done as follows:

```
if [ -z "$1" ]; then
    echo "No argument supplied"
fi
```

Getting all the parameters

`$@` and `$*` are ways of interacting with all the script parameters. Referencing the Bash man page, we see that:

- `$*`: Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable.
- `$@`: Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word.

Getting the number of parameters

`$#` gets the number of parameters passed into a script. A typical use case would be to check if the appropriate number of arguments are passed:

```
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
fi
```

Example 1

Loop through all arguments and check if they are files:

```
for item in "$@"
do
    if [[ -f $item ]]; then
        echo "$item is a file"
```

```
    fi
done
```

Example 2

Loop through all arguments and check if they are files:

```
for (( i = 1; i <= $#; ++ i ))
do
    item=${@:$i:1}

    if [[ -f $item ]]; then
        echo "$item is a file"
    fi
done
```

Section 23.5: Split string into an array in Bash

Let's say we have a String parameter and we want to split it by comma

```
my_param="foo,bar,bash"
```

To split this string by comma we can use;

```
IFS=',' read -r -a array <<< "$my_param"
```

Here, IFS is a special variable called Internal field separator which defines the character or characters used to separate a pattern into tokens for some operations.

To access an individual element:

```
echo "${array[0]}"
```

To iterate over the elements:

```
for element in "${array[@]}"
do
    echo "$element"
done
```

To get both the index and the value:

```
for index in "${!array[@]}"
do
    echo "$index ${array[index]}"
done
```

Chapter 24: Bash history substitutions

Section 24.1: Quick Reference

Interaction with the history

```
# List all previous commands
history

# Clear the history, useful if you entered a password by accident
history -c
```

Event designators

```
# Expands to line n of bash history
!n

# Expands to last command
!!

# Expands to last command starting with "text"
!text

# Expands to last command containing "text"
!?text

# Expands to command n lines ago
!-n

# Expands to last command with first occurrence of "foo" replaced by "bar"
^foo^bar^

# Expands to the current command
!#
```

Word designators

These are separated by `:` from the event designator they refer to. The colon can be omitted if the word designator doesn't start with a number: `!^` is the same as `!:^`.

```
# Expands to the first argument of the most recent command
!^

# Expands to the last argument of the most recent command (short for !!:$)
!$

# Expands to the third argument of the most recent command
!:3

# Expands to arguments x through y (inclusive) of the last command
# x and y can be numbers or the anchor characters ^ $
!:x-y

# Expands to all words of the last command except the 0th
# Equivalent to :^-$
!*
```

Modifiers

These modify the preceding event or word designator.

```
# Replacement in the expansion using sed syntax
```

```
# Allows flags before the s and alternate separators
:s/foo/bar/ #substitutes bar for first occurrence of foo
:gs|foo|bar| #substitutes bar for all foo

# Remove leading path from last argument ("tail")
:t

# Remove trailing path from last argument ("head")
:h

# Remove file extension from last argument
:r
```

If the Bash variable `HISTCONTROL` contains either `ignorespace` or `ignoreboth` (or, alternatively, `HISTIGNORE` contains the pattern `[ ]*`), you can prevent your commands from being stored in Bash history by prepending them with a space:

```
# This command won't be saved in the history
 foo

# This command will be saved
bar
```

Section 24.2: Repeat previous command with sudo

```
$ apt-get install r-base
E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)
E: Unable to lock the administration directory (/var/lib/dpkg/), are you root?
$ sudo !!
sudo apt-get install r-base
[sudo] password for <user>:
```

Section 24.3: Search in the command history by pattern

Press `control` `r` and type a pattern.

For example, if you recently executed `man 5 crontab`, you can find it quickly by *starting to type* "crontab". The prompt will change like this:

```
(reverse-i-search)`cr': man 5 crontab
```

The `` `cr' `` there is the string I typed so far. This is an incremental search, so as you continue typing, the search result gets updated to match the most recent command that contained the pattern.

Press the left or right arrow keys to edit the matched command before running it, or the `enter` key to run the command.

By default the search finds the most recently executed command matching the pattern. To go further back in the history press `control` `r` again. You may press it repeatedly until you find the desired command.

Section 24.4: Switch to newly created directory with !#:N

```
$ mkdir backup_download_directory && cd !#:1
mkdir backup_download_directory && cd backup_download_directory
```

This will substitute the Nth argument of the current command. In the example `!#:1` is replaced with the first

argument, i.e. backup_download_directory.

Section 24.5: Using !$

You can use the `!$` to reduce repetition when using the command line:

```
$ echo ping
ping
$ echo !$
ping
```

You can also build upon the repetition

```
$ echo !$ pong
ping pong
$ echo !$, a great game
pong, a great game
```

Notice that in the last example we did not get `ping pong, a great game` because the last argument passed to the previous command was `pong`, we can avoid issue like this by adding quotes. Continuing with the example, our last argument was `game`:

```
$ echo "it is !$ time"
it is game time
$ echo "hooray, !$!"
hooray, it is game time!
```

Section 24.6: Repeat the previous command with a substitution

```
$ mplayer Lecture_video_part1.mkv
$ ^1^2^
mplayer Lecture_video_part2.mkv
```

This command will replace `1` with `2` in the previously executed command. It will only replace the first occurrence of the string and is equivalent to `!!:s/1/2/`.

If you want to replace *all* occurrences, you have to use `!!:gs/1/2/` or `!!:as/1/2/`.

Chapter 25: Math

Section 25.1: Math using dc

`dc` is one of the oldest programs on Unix.

It uses reverse polish notation, which means that you first stack numbers, then operations. For example `1+1` is written as `1 1+`.

To print an element from the top of the stack use command `p`

```
echo '2 3 + p' | dc
5

or

dc <<< '2 3 + p'
5
```

You can print the top element many times

```
dc <<< '1 1 + p 2 + p'
2
4
```

For negative numbers use `_` prefix

```
dc <<< '_1 p'
-1
```

You can also use capital letters from `A to F` for numbers between `10 and 15` and `.` as a decimal point

```
dc <<< 'A.4 p'
10.4
```

`dc` is using abitrary precision which means that the precision is limited only by the available memory. By default the precision is set to 0 decimals

```
dc <<< '4 3 / p'
1
```

We can increase the precision using command `k`. `2k` will use

```
dc <<< '2k 4 3 / p'
1.33

dc <<< '4k 4 3 / p'
1.3333
```

You can also use it over multiple lines

```
dc << EOF
1 1 +
3 *
p
```

```
EOF
6
```

> bc is a preprocessor for dc.

Section 25.2: Math using bash capabilities

Arithmetic computation can be also done without involving any other programs like this:

Multiplication:

```
echo $((5 * 2))
10
```

Division:

```
echo $((5 / 2))
2
```

Modulo:

```
echo $((5 % 2))
1
```

Exponentiation:

```
echo $((5 ** 2))
25
```

Section 25.3: Math using bc

bc is an arbitrary precision calculator language. It could be used interactively or be executed from command line.

For example, it can print out the result of an expression:

```
echo '2 + 3' | bc
5

echo '12 / 5' | bc
2
```

For floating-post arithmetic, you can import standard library `bc -l`:

```
echo '12 / 5' | bc -l
2.40000000000000000000
```

It can be used for comparing expressions:

```
echo '8 > 5' | bc
1

echo '10 == 11' | bc
0
```

```
echo '10 == 10 && 8 > 3' | bc
1
```

Section 25.4: Math using expr

`expr` or `Evaluate expressions` evaluates an expression and writes the result on standard output

Basic arithmetics

```
expr 2 + 3
5
```

When multiplying, you need to escape the `*` sign

```
expr 2 \* 3
6
```

You can also use variables

```
a=2
expr $a + 3
5
```

Keep in mind that it only supports integers, so expression like this

```
expr 3.0 / 2
```

will throw an error `expr: not a decimal number: '3.0'`.

It supports regular expression to match patterns

```
expr 'Hello World' : 'Hell\(.*\)rld'
o Wo
```

Or find the index of the first char in the search string

> This will throw `expr: syntax error` on **Mac OS X**, because it uses **BSD expr** which does not have the index command, while expr on Linux is generally **GNU expr**

```
expr index hello l
3

expr index 'hello' 'lo'
3
```

Chapter 26: Bash Arithmetic

Parameter	Details
EXPRESSION	Expression to evaluate

Section 26.1: Simple arithmetic with (( ))

```
#!/bin/bash
echo $(( 1 + 2 ))
```

Output: 3

```
# Using variables
#!/bin/bash
var1=4
var2=5
((output=$var1 * $var2))
printf "%d\n" "$output"
```

Output: 20

Section 26.2: Arithmetic command

- `let`

```
    let num=1+2
    let num="1+2"
    let 'num= 1 + 2'
    let num=1 num+=2
```

You need quotes if there are spaces or globbing characters. So those will get error:

```
   let num = 1 + 2    #wrong
   let 'num = 1 + 2'  #right
   let a[1] = 1 + 1   #wrong
   let 'a[1] = 1 + 1' #right
```

- `(( ))`

```
  ((a=$a+1))     #add 1 to a
  ((a = a + 1))  #like above
  ((a += 1))     #like above
```

We can use `(())` in `if`. Some Example:

```
if (( a > 1 )); then echo "a is greater than 1"; fi
```

The output of `(())` can be assigned to a variable:

```
result=$((a + 1))
```

Or used directly in output:

```
echo "The result of a + 1 is $((a + 1))"
```

Section 26.3: Simple arithmetic with expr

```
#!/bin/bash
expr 1 + 2
```

Output: 3

Chapter 27: Scoping

Section 27.1: Dynamic scoping in action

Dynamic scoping means that variable lookups occur in the scope where a function is *called*, not where it is *defined*.

```
$ x=3
$ func1 () { echo "in func1: $x"; }
$ func2 () { local x=9; func1; }
$ func2
in func1: 9
$ func1
in func1: 3
```

In a lexically scoped language, `func1` would *always* look in the global scope for the value of `x`, because `func1` is *defined* in the local scope.

In a dynamically scoped language, `func1` looks in the scope where it is *called*. When it is called from within `func2`, it first looks in the body of `func2` for a value of `x`. If it weren't defined there, it would look in the global scope, where `func2` was called from.

Chapter 28: Process substitution

Section 28.1: Compare two files from the web

The following compares two files with `diff` using process substitution instead of creating temporary files.

```
diff <(curl http://www.example.com/page1) <(curl http://www.example.com/page2)
```

Section 28.2: Feed a while loop with the output of a command

This feeds a `while` loop with the output of a `grep` command:

```
while IFS=":" read -r user _
do
    # "$user" holds the username in /etc/passwd
done < <(grep "hello" /etc/passwd)
```

Section 28.3: Concatenating files

It is well known that you cannot use the same file for input and output in the same command. For instance,

```
$ cat header.txt body.txt >body.txt
```

doesn't do what you want. By the time `cat` reads `body.txt`, it has already been truncated by the redirection and it is empty. The final result will be that `body.txt` will hold the contents of `header.txt` *only.*

One might think to avoid this with process substitution, that is, that the command

```
$ cat header.txt <(cat body.txt) > body.txt
```

will force the original contents of `body.txt` to be somehow saved in some buffer somewhere before the file is truncated by the redirection. It doesn't work. The `cat` in parentheses begins reading the file only after all file descriptors have been set up, just like the outer one. There is no point in trying to use process substitution in this case.

The only way to prepend a file to another file is to create an intermediate one:

```
$ cat header.txt body.txt >body.txt.new
$ mv body.txt.new body.txt
```

which is what `sed` or `perl` or similar programs do under the carpet when called with an *edit-in-place* option (usually `-i`).

Section 28.4: Stream a file through multiple programs at once

This counts the number of lines in a big file with `wc -l` while simultaneously compressing it with `gzip`. Both run concurrently.

```
tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Normally `tee` writes its input to one or more files (and stdout). We can write to commands instead of files with `tee`

`>(command)`.

Here the command `wc -l >&2` counts the lines read from `tee` (which in turn is reading from `bigfile`). (The line count is sent to stderr (`>&2`) to avoid mixing with the input to `gzip`.) The stdout of `tee` is simultaneously fed into `gzip`.

Section 28.5: With paste command

```
# Process substitution with paste command is common
# To compare the contents of two directories
paste <( ls /path/to/directory1 ) <( ls /path/to/directory2 )
```

Section 28.6: To avoid usage of a sub-shell

One major aspect of process substitution is that it lets us avoid usage of a sub-shell when piping commands from the shell.

This can be demonstrated with a simple example below. I have the following files in my current folder:

```
$ find . -maxdepth 1 -type f -print
foo bar zoo foobar foozoo barzoo
```

If I pipe to a `while`/`read` loop that increments a counter as follows:

```
count=0
find . -maxdepth 1 -type f -print | while IFS= read -r _; do
    ((count++))
done
```

`$count` now does *not* contain `6`, because it was modified in the sub-shell context. Any of the commands shown below are run in a sub-shell context and the scope of the variables used within are lost after the sub-shell terminates.

```
command &
command | command
( command )
```

Process substitution will solve the problem by avoiding use the of pipe `|` operator as in

```
count=0
while IFS= read -r _; do
    ((count++))
done < <(find . -maxdepth 1 -type f -print)
```

This will retain the `count` variable value as no sub-shells are invoked.

Chapter 29: Programmable completion

Section 29.1: Simple completion using function

```
_mycompletion() {
    local command_name="$1" # not used in this example
    local current_word="$2"
    local previous_word="$3" # not used in this example
    # COMPREPLY is an array which has to be filled with the possible completions
    # compgen is used to filter matching completions
    COMPREPLY=( $(compgen -W 'hello world' -- "$current_word") )
}
complete -F _mycompletion mycommand
```

Usage Example:

```
$ mycommand [TAB][TAB]
hello world
$ mycommand h[TAB][TAB]
$ mycommand hello
```

Section 29.2: Simple completion for options and filenames

```
# The following shell function will be used to generate completions for
# the "nuance_tune" command.
_nuance_tune_opts ()
{
  local curr_arg prev_arg
  curr_arg=${COMP_WORDS[COMP_CWORD]}
  prev_arg=${COMP_WORDS[COMP_CWORD-1]}

  # The "config" option takes a file arg, so get a list of the files in the
  # current dir.  A case statement is probably unnecessary here, but leaves
  # room to customize the parameters for other flags.
  case "$prev_arg" in
    -config)
      COMPREPLY=( $( /bin/ls -1 ) )
      return 0
      ;;
    esac

    # Use compgen to provide completions for all known options.
    COMPREPLY=( $(compgen -W '-analyze -experiment -generate_groups -compute_thresh -config -output
-help -usage -force -lang -grammar_overrides -begin_date -end_date -group -dataset -multiparses -
dump_records -no_index -confidencelevel -nrecs -dry_run -rec_scripts_only -save_temp -full_trc -
single_session -verbose -ep -unsupervised -write_manifest -remap -noreparse -upload -reference -
target -use_only_matching -histogram -stepsize' -- $curr_arg ) );
}

# The -o parameter tells Bash to process completions as filenames, where applicable.


complete -o filenames -F _nuance_tune_opts nuance_tune
```

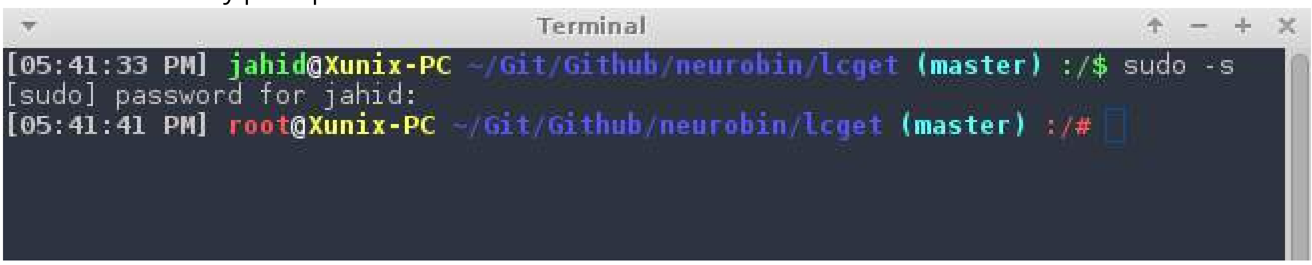
Chapter 30: Customizing PS1

Section 30.1: Colorize and customize terminal prompt

This is how the author sets their personal PS1 variable:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}
#Please use the below function if you are a mac user
gitPS1ForMac(){
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
timeNow(){
    echo "$(date +%r)"
}
if [ "$color_prompt" = yes ]; then
  if [ x$EUID = x0 ]; then
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;31m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;31m\]:/#\[\033[00m\] '
  else
    PS1='\[\033[1;38m\][$(timeNow)]\[\033[00m\]
\[\033[1;32m\]\u\[\033[00m\]\[\033[1;37m\]@\[\033[00m\]\[\033[1;33m\]\h\[\033[00m\]
\[\033[1;34m\]\w\[\033[00m\]\[\033[1;36m\]$(gitPS1)\[\033[00m\] \[\033[1;32m\]:/$\[\033[00m\] '
  fi
else
    PS1='[$(timeNow)] \u@\h \w$(gitPS1) :/$ '
fi
```

And this is how my prompt looks like:



Color reference:

```
# Colors
txtblk='\e[0;30m' # Black - Regular
txtred='\e[0;31m' # Red
txtgrn='\e[0;32m' # Green
txtylw='\e[0;33m' # Yellow
txtblu='\e[0;34m' # Blue
txtpur='\e[0;35m' # Purple
txtcyn='\e[0;36m' # Cyan
txtwht='\e[0;37m' # White
bldblk='\e[1;30m' # Black - Bold
bldred='\e[1;31m' # Red
bldgrn='\e[1;32m' # Green
bldylw='\e[1;33m' # Yellow
bldblu='\e[1;34m' # Blue
bldpur='\e[1;35m' # Purple
bldcyn='\e[1;36m' # Cyan
```

```
bldwht='\e[1;37m' # White
unkblk='\e[4;30m' # Black - Underline
undred='\e[4;31m' # Red
undgrn='\e[4;32m' # Green
undylw='\e[4;33m' # Yellow
undblu='\e[4;34m' # Blue
undpur='\e[4;35m' # Purple
undcyn='\e[4;36m' # Cyan
undwht='\e[4;37m' # White
bakblk='\e[40m'   # Black - Background
bakred='\e[41m'   # Red
badgrn='\e[42m'   # Green
bakylw='\e[43m'   # Yellow
bakblu='\e[44m'   # Blue
bakpur='\e[45m'   # Purple
bakcyn='\e[46m'   # Cyan
bakwht='\e[47m'   # White
txtrst='\e[0m'    # Text Reset
```

Notes:

- Make the changes in `~/.bashrc` or `/etc/bashrc` or `~/.bash_profile` or `~./profile` file (depending on the OS) and save it.
- For `root` you might also need to edit the `/etc/bash.bashrc` or `/root/.bashrc` file
- Run `source ~/.bashrc` (distro specific) after saving the file.
- Note: if you have saved the changes in `~/.bashrc`, then remember to add `source ~/.bashrc` in your `~/.bash_profile` so that this change in `PS1` will be recorded every time the Terminal application starts.

Section 30.2: Show git branch name in terminal prompt

You can have functions in the PS1 variable, just make sure to single quote it or use escape for special chars:

```
gitPS1(){
    gitps1=$(git branch 2>/dev/null | grep '*')
    gitps1="${gitps1:+ (${gitps1/#\* /})}"
    echo "$gitps1"
}

PS1='\u@\h:\w$(gitPS1)$ '
```

It will give you a prompt like this:

```
user@Host:/path (master)$
```

Notes:

- Make the changes in `~/.bashrc` or `/etc/bashrc` or `~/.bash_profile` or `~./profile` file (depending on the OS) and save it.
- Run `source ~/.bashrc` (distro specific) after saving the file.

Section 30.3: Show time in terminal prompt

```
timeNow(){
    echo "$(date +%r)"
}
```

```
PS1='[$(timeNow)] \u@\h:\w$ '
```

It will give you a prompt like this:

```
[05:34:37 PM] user@Host:/path$
```

Notes:

- Make the changes in `~/.bashrc` or `/etc/bashrc` or `~/.bash_profile` or `~./profile` file (depending on the OS) and save it.
- Run `source ~/.bashrc` (distro specific) after saving the file.

Section 30.4: Show a git branch using PROMPT_COMMAND

If you are inside a folder of a git repository it might be nice to show the current branch you are on. In `~/.bashrc` or `/etc/bashrc` add the following (git is required for this to work):

```
function prompt_command {
    # Check if we are inside a git repository
    if git status > /dev/null 2>&1; then
        # Only get the name of the branch
        export GIT_STATUS=$(git status | grep 'On branch' | cut -b 10-)
    else
        export GIT_STATUS=""
    fi
}
# This function gets called every time PS1 is shown
PROMPT_COMMAND=prompt_command

PS1="\$GIT_STATUS \u@\h:\w\$ "
```

If we are in a folder inside a git repository this will output:

> branch user@machine:~$

And if we are inside a normal folder:

> user@machine:~$

Section 30.5: Change PS1 prompt

To change PS1, you just have to change the value of PS1 shell variable. The value can be set in `~/.bashrc` or `/etc/bashrc` file, depending on the distro. PS1 can be changed to any plain text like:

```
PS1="hello "
```

Besides the plain text, a number of backslash-escaped special characters are supported:

Format	Action
`\a`	an ASCII bell character (07)
`\d`	the date in “Weekday Month Date” format (e.g., “Tue May 26”)

`\D{format}`	the format is passed to strftime(3) and the result is inserted into the prompt string; an empty format results in a locale-specific time representation. The braces are required
`\e`	an ASCII escape character (033)
`\h`	the hostname up to the first '.'
`\H`	the hostname
`\j`	the number of jobs currently managed by the shell
`\l`	the basename of the shell's terminal device name
`\n`	newline
`\r`	carriage return
`\s`	the name of the shell, the basename of $0 (the portion following the final slash)
`\t`	the current time in 24-hour HH:MM:SS format
`\T`	the current time in 12-hour HH:MM:SS format
`\@`	the current time in 12-hour am/pm format
`\A`	the current time in 24-hour HH:MM format
`\u`	the username of the current user
`\v`	the version of bash (e.g., 2.00)
`\V`	the release of bash, version + patch level (e.g., 2.00.0)
`\w`	the current working directory, with $HOME abbreviated with a tilde
`\W`	the basename of the current working directory, with $HOME abbreviated with a tilde
`\!`	the history number of this command
`\#`	the command number of this command
`\$`	if the effective UID is 0, a #, otherwise a $
`\nnn*`	the character corresponding to the octal number nnn
`\`	a backslash
`\[`	begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
`\]`	end a sequence of non-printing characters

So for example, we can set PS1 to:

```
PS1="\u@\h:\w\$ "
```

And it will output:

> user@machine:~$

Section 30.6: Show previous command return status and time

Sometimes we need a visual hint to indicate the return status of previous command. The following snippet make put it at the head of the PS1.

Note that the __stat() function should be called every time a new PS1 is generated, or else it would stick to the return status of last command of your .bashrc or .bash_profile.

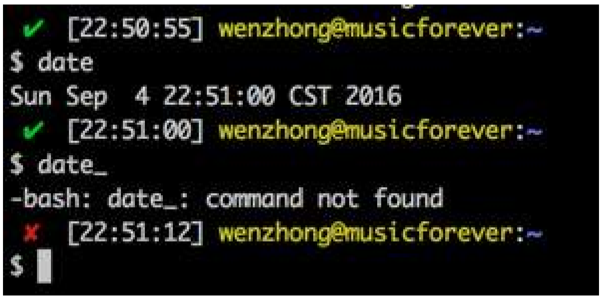
```
# -ANSI-COLOR-CODES- #
Color_Off="\033[0m"
###-Regular-###
Red="\033[0;31m"
Green="\033[0;32m"
```

```
Yellow="\033[0;33m"
####-Bold-####

function __stat() {
    if [ $? -eq 0 ]; then
        echo -en "$Green ✔ $Color_Off "
    else
        echo -en "$Red ✘ $Color_Off "
    fi
}

PS1='$(__stat)'
PS1+="[\t] "
PS1+="\e[0;33m\u@\h\e[0m:\e[1;34m\w\e[0m \n$ "

export PS1
```



Chapter 31: Brace Expansion

Section 31.1: Modifying filename extension

```
$ mv filename.{jar,zip}
```

This expands into `mv filename.jar filename.zip` .

Section 31.2: Create directories to group files by month and year

```
$ mkdir 20{09..11}-{01..12}
```

Entering the `ls` command will show that the following directories were created:

```
2009-01 2009-04 2009-07 2009-10 2010-01 2010-04 2010-07 2010-10 2011-01 2011-04 2011-07 2011-10
2009-02 2009-05 2009-08 2009-11 2010-02 2010-05 2010-08 2010-11 2011-02 2011-05 2011-08 2011-11
2009-03 2009-06 2009-09 2009-12 2010-03 2010-06 2010-09 2010-12 2011-03 2011-06 2011-09 2011-12
```

Putting a `0` in front of `9` in the example ensures the numbers are padded with a single `0`. You can also pad numbers with multiple zeros, for example:

```
$ echo {001..10}
001 002 003 004 005 006 007 008 009 010
```

Section 31.3: Create a backup of dotfiles

```
$ cp .vimrc{,.bak}
```

This expands into the command `cp .vimrc .vimrc.bak`.

Section 31.4: Use increments

```
$ echo {0..10..2}
0 2 4 6 8 10
```

A third parameter to specify an increment, i.e. `{start..end..increment}`

Using increments is not constrained to just numbers

```
$ for c in {a..z..5}; do echo -n $c; done
afkpuz
```

Section 31.5: Using brace expansion to create lists

Bash can easily create lists from alphanumeric characters.

```
# list from a to z
$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z

# reverse from z to a
$ echo {z..a}
```

```
z y x w v u t s r q p o n m l k j i h g f e d c b a

# digits
$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

# with leading zeros
$ echo {01..20}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

# reverse digit
$ echo {20..1}
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# reversed with leading zeros
$ echo {20..01}
20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01

# combining multiple braces
$ echo {a..d}{1..3}
a1 a2 a3 b1 b2 b3 c1 c2 c3 d1 d2 d3
```

Brace expansion is the very first expansion that takes place, so it cannot be combined with any other expansions.

Only chars and digits can be used.

This won't work: `echo {$(date +$H)..24}`

Section 31.6: Make Multiple Directories with Sub-Directories

```
mkdir -p toplevel/sublevel_{01..09}/{child1,child2,child3}
```

This will create a top level folder called `toplevel`, nine folders inside of `toplevel` named `sublevel_01`, `sublevel_02`, etc. Then inside of those sublevels: `child1`, `child2`, `child3` folders, giving you:

```
toplevel/sublevel_01/child1
toplevel/sublevel_01/child2
toplevel/sublevel_01/child3
toplevel/sublevel_02/child1
```

and so on. I find this very useful for creating multiple folders and sub folders for my specific purposes, with one bash command. Substitute variables to help automate/parse information given to the script.

Chapter 32: getopts : smart positional-parameter parsing

Parameter	Detail
optstring	The option characters to be recognized
name	Then name where parsed option is stored

Section 32.1: pingnmap

```
#!/bin/bash
# Script name : pingnmap
# Scenario : The systems admin in company X is tired of the monotonous job
# of pinging and nmapping, so he decided to simplify the job using a script.
# The tasks he wish to achieve is
# 1. Ping - with a max count of 5 -the given IP address/domain. AND/OR
# 2. Check if a particular port is open with a given IP address/domain.
# And getopts is for her rescue.
# A brief overview of the options
# n : meant for nmap
# t : meant for ping
# i : The option to enter the IP address
# p : The option to enter the port
# v : The option to get the script version


while getopts ':nti:p:v' opt
#putting : in the beginnnig suppresses the errors for invalid options
do
case "$opt" in
   'i')ip="${OPTARG}"
       ;;
   'p')port="${OPTARG}"
       ;;
   'n')nmap_yes=1;
       ;;
   't')ping_yes=1;
       ;;
   'v')echo "pingnmap version 1.0.0"
       ;;
    *) echo "Invalid option $opt"
       echo "Usage : "
       echo "pingmap -[n|t[i|p]|v]"
       ;;
esac
done
if  [ ! -z "$nmap_yes" ] && [ "$nmap_yes" -eq "1" ]
then
   if [ ! -z "$ip" ] && [ ! -z "$port" ]
   then
     nmap -p "$port" "$ip"
   fi
fi

if  [ ! -z "$ping_yes" ] && [ "$ping_yes" -eq "1" ]
then
   if [ ! -z "$ip" ]
   then
     ping -c 5 "$ip"
```

```
    fi
fi
shift $(( OPTIND - 1 )) # Processing additional arguments
if [ ! -z "$@" ]
then
  echo "Bogus arguments at the end : $@"
fi
```

Output

```
$ ./pingnmap -nt -i google.com -p 80

Starting Nmap 6.40 ( http://nmap.org ) at 2016-07-23 14:31 IST
Nmap scan report for google.com (216.58.197.78)
Host is up (0.034s latency).
rDNS record for 216.58.197.78: maa03s21-in-f14.1e100.net
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.22 seconds
PING google.com (216.58.197.78) 56(84) bytes of data.
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=1 ttl=57 time=29.3 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=2 ttl=57 time=30.9 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=3 ttl=57 time=34.7 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=4 ttl=57 time=39.6 ms
64 bytes from maa03s21-in-f14.1e100.net (216.58.197.78): icmp_seq=5 ttl=57 time=32.7 ms

--- google.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 29.342/33.481/39.631/3.576 ms
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
$ ./pingnmap -v
pingnmap version 1.0.0
$ ./pingnmap -h
Invalid option ?
Usage :
pingmap -[n|t[i|p]|v]
```

Chapter 33: Debugging

Section 33.1: Checking the syntax of a script with "-n"

The -n flag enables you to check the syntax of a script without having to execute it:

```
~> $ bash -n testscript.sh
testscript.sh: line 128: unexpected EOF while looking for matching `"'
testscript.sh: line 130: syntax error: unexpected end of file
```

Section 33.2: Debugging using bashdb

Bashdb is a utility that is similar to gdb, in that you can do things like set breakpoints at a line or at a function, print content of variables, you can restart script execution and more.

You can normally install it via your package manager, for example on Fedora:

```
sudo dnf install bashdb
```

Or get it from the homepage. Then you can run it with your script as a paramater:

```
bashdb <YOUR SCRIPT>
```

Here are a few commands to get you started:

```

l - show local lines, press l again to scroll down
s - step to next line
print $VAR - echo out content of variable
restart - reruns bashscript, it re-loads it prior to execution.
eval - evaluate some custom command, ex: eval echo hi

b  set breakpoint on some line
c - continue till some breakpoint
i b - info on break points
d  - delete breakpoint at line #

shell - launch a sub-shell in the middle of execution, this is handy for manipulating variables
```

For more information, I recommend consulting the manual:
http://www.rodericksmith.plus.com/outlines/manuals/bashdbOutline.html

See also homepage:
http://bashdb.sourceforge.net/

Section 33.3: Debugging a bash script with "-x"

Use "-x" to enable debug output of executed lines. It can be run on an entire session or script, or enabled programmatically within a script.

Run a script with debug output enabled:

```
$ bash -x myscript.sh
```

Or

```
$ bash --debug myscript.sh
```

Turn on debugging within a bash script. It may optionally be turned back on, though debug output is automatically reset when the script exits.

```
#!/bin/bash
set -x   # Enable debugging
# some code here
set +x   # Disable debugging output.
```

Chapter 34: Pattern matching and regular expressions

Section 34.1: Get captured groups from a regex match against a string

```
a='I am a simple string with digits 1234'
pat='(.*) ([0-9]+)'
[[ "$a" =~ $pat ]]
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
echo "${BASH_REMATCH[2]}"
```

Output:

```
I am a simple string with digits 1234
I am a simple string with digits
1234
```

Section 34.2: Behaviour when a glob does not match anything

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

In case the glob does not match anything the result is determined by the options `nullglob` and `failglob`. If neither of them are set, Bash will return the glob itself if nothing is matched

```
$ echo no*match
no*match
```

If `nullglob` is activated then nothing (`null`) is returned:

```
$ shopt -s nullglob
$ echo no*match

$
```

If `failglob` is activated then an error message is returned:

```
$ shopt -s failglob
$ echo no*match
bash: no match: no*match
$
```

Notice, that the `failglob` option supersedes the `nullglob` option, i.e., if `nullglob` and `failglob` are both set, then - in case of no match - an error is returned.

Section 34.3: Check if a string matches a regular expression

Version ≥ 3.0

Check if a string consists in exactly 8 digits:

```
$ date=20150624
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
yes
$ date=hello
$ [[ $date =~ ^[0-9]{8}$ ]] && echo "yes" || echo "no"
no
```

Section 34.4: Regex matching

```
pat='[^0-9]+([0-9]+)'
s='I am a string with some digits 1024'
[[ $s =~ $pat ]] # $pat must be unquoted
echo "${BASH_REMATCH[0]}"
echo "${BASH_REMATCH[1]}"
```

Output:

```
I am a string with some digits 1024
1024
```

Instead of assigning the regex to a variable (`$pat`) we could also do:

```
[[ $s =~ [^0-9]+([0-9]+) ]]
```

Explanation

- The `[[ $s =~ $pat ]]` construct performs the regex matching
- The captured groups i.e the match results are available in an array named *BASH_REMATCH*
- The 0th index in the *BASH_REMATCH* array is the total match
- The *i'th* index in the *BASH_REMATCH* array is the *i'th* captured group, where *i* = 1, 2, 3 ...

Section 34.5: The * glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

The asterisk * is probably the most commonly used glob. It simply matches any String

```
$ echo *acy
macy stacy tracy
```

A single * will not match files and folders that reside in subfolders

```
$ echo *
emptyfolder folder macy stacy tracy
$ echo folder/*
folder/anotherfolder folder/subfolder
```

Section 34.6: The ** glob

Version ≥ 4.0

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -s globstar
```

Bash is able to interpret two adjacent asterisks as a single glob. With the `globstar` option activated this can be used to match folders that reside deeper in the directory structure

```
echo **
emptyfolder folder folder/anotherfolder folder/anotherfolder/content
folder/anotherfolder/content/deepfolder folder/anotherfolder/content/deepfolder/file
folder/subfolder folder/subfolder/content folder/subfolder/content/deepfolder
folder/subfolder/content/deepfolder/file macy stacy tracy
```

The `**` can be thought of a path expansion, no matter how deep the path is. This example matches any file or folder that starts with `deep`, regardless of how deep it is nested:

```
$ echo **/deep*
folder/anotherfolder/content/deepfolder folder/subfolder/content/deepfolder
```

Section 34.7: The ? glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
```

```
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

The `?` simply matches exactly one character

```
$ echo ?acy
macy
$ echo ??acy
stacy tracy
```

Section 34.8: The [ ] glob

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

If there is a need to match specific characters then '[]' can be used. Any character inside '[]' will be matched exactly once.

```
$ echo [m]acy
macy
$ echo [st][tr]acy
stacy tracy
```

The `[]` glob, however, is more versatile than just that. It also allows for a negative match and even matching ranges of characters and character classes. A negative match is achieved by using `!` or `^` as the first character following `[`. We can match `stacy` by

```
$ echo [!t][^r]acy
stacy
```

Here we are telling bash the we want to match only files which do not not start with a `t` and the second letter is not an `r` and the file ends in `acy`.

Ranges can be matched by seperating a pair of characters with a hyphen (`-`). Any character that falls between those two enclosing characters - inclusive - will be matched. E.g., `[r-t]` is equivalent to `[rst]`

```
$ echo [r-t][r-t]acy
stacy tracy
```

Character classes can be matched by `[:class:]`, e.g., in order to match files that contain a whitespace

```
$ echo *[[:blank:]]*
file with space
```

Section 34.9: Matching hidden files

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

The Bash built-in option *dotglob* allows to match hidden files and folders, i.e., files and folders that start with a `.`

```
$ shopt -s dotglob
$ echo *
file with space folder .hiddenfile macy stacy tracy
```

Section 34.10: Case insensitive matching

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Setting the option `nocaseglob` will match the glob in a case insensitive manner

```
$ echo M*
M*
$ shopt -s nocaseglob
$ echo M*
macy
```

Section 34.11: Extended globbing

Version ≥ 2.02

Preparation

```
$ mkdir globbing
$ cd globbing
$ mkdir -p folder/{sub,another}folder/content/deepfolder/
touch macy stacy tracy "file with space" folder/{sub,another}folder/content/deepfolder/file
.hiddenfile
$ shopt -u nullglob
```

```
$ shopt -u failglob
$ shopt -u dotglob
$ shopt -u nocaseglob
$ shopt -u extglob
$ shopt -u globstar
```

Bash's built-in `extglob` option can extend a glob's matching capabilities

```
shopt -s extglob
```

The following sub-patterns comprise valid extended globs:

- `?(pattern-list)` – Matches zero or one occurrence of the given patterns
- `*(pattern-list)` – Matches zero or more occurrences of the given patterns
- `+(pattern-list)` – Matches one or more occurrences of the given patterns
- `@(pattern-list)` – Matches one of the given patterns
- `!(pattern-list)` – Matches anything except one of the given patterns

The `pattern-list` is a list of globs separated by `|`.

```
$ echo *([r-t])acy
stacy tracy

$ echo *([r-t]|m)acy
macy stacy tracy

$ echo ?([a-z])acy
macy
```

The `pattern-list` itself can be another, nested extended glob. In the above example we have seen that we can match `tracy` and `stacy` with `*(r-t)`. This extended glob itself can be used inside the negated extended glob `!(pattern-list)` in order to match `macy`

```
$ echo !(*([r-t]))acy
macy
```

It matches anything that does **not** start with zero or more occurrences of the letters `r`, `s` and `t`, which leaves only `macy` as possible match.

Chapter 35: Change shell

Section 35.1: Find the current shell

There are a few ways to determine the current shell

```
echo $0
ps -p $$
echo $SHELL
```

Section 35.2: List available shells

To list available login shells :

```
cat /etc/shells
```

Example:

```
$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
```

Section 35.3: Change the shell

To change the current bash run these commands

```
export SHELL=/bin/bash
exec /bin/bash
```

to change the bash that opens on startup edit `.profile` and add those lines

Chapter 36: Internal variables

An overview of Bash's internal variables, where, how, and when to use them.

Section 36.1: Bash internal variables at a glance

Variable	Details
`$*` / `$@`	Function/script positional parameters (arguments). Expand as follows: `$*` and `$@` are the same as `$1 $2 ...` (note that it generally makes no sense to leave those unquoted) `"$*"` is the same as `"$1 $2 ..."` 1 `"$@"` is the same as `"$1" "$2" ...` 1. Arguments are separated by the first character of $IFS, which does not have to be a space.
`$#`	Number of positional parameters passed to the script or function
`$!`	Process ID of the last (righ-most for pipelines) command in the most recently job put into the background (note that it's not necessarily the same as the job's process group ID when job control is enabled)
`$$`	ID of the process that executed `bash`
`$?`	Exit status of the last command
`$n`	Positional parameters, where n=1, 2, 3, ..., 9
`${n}`	Positional parameters (same as above), but n can be > 9
`$0`	In scripts, path with which the script was invoked; with `bash -c 'printf "%s\n" "$0"' name args'`: `name` (the first argument after the inline script), otherwise, the `argv[0]` that `bash` received.
`$_`	Last field of the last command
`$IFS`	Internal field separator
`$PATH`	PATH environment variable used to look-up executables
`$OLDPWD`	Previous working directory
`$PWD`	Present working directory
`$FUNCNAME`	Array of function names in the execution call stack
`$BASH_SOURCE`	Array containing source paths for elements in `FUNCNAME` array. Can be used to get the script path.
`$BASH_ALIASES`	Associative array containing all currently defined aliases
`$BASH_REMATCH`	Array of matches from the last regex match
`$BASH_VERSION`	Bash version string
`$BASH_VERSINFO`	An array of 6 elements with Bash version information
`$BASH`	Absolute path to the currently executing Bash shell itself (heuristically determined by `bash` based on `argv[0]` and the value of `$PATH`; may be wrong in corner cases)
`$BASH_SUBSHELL`	Bash subshell level
`$UID`	Real (not effective if different) User ID of the process running `bash`
`$PS1`	Primary command line prompt; see Using the PS* Variables
`$PS2`	Secondary command line prompt (used for additional input)
`$PS3`	Tertiary command line prompt (used in select loop)
`$PS4`	Quaternary command line prompt (used to append info with verbose output)
`$RANDOM`	A pseudo random integer between 0 and 32767
`$REPLY`	Variable used by `read` by default when no variable is specified. Also used by `SELECT` to return the user-supplied value
`$PIPESTATUS`	Array variable that holds the exit status values of each command in the most recently executed foreground pipeline.

> Variable Assignment must have no space before and after. a=123 not a = 123. The latter (an equal sign surrounded by spaces) in isolation means run the command a with the arguments = and 123, though it is also seen in the string comparison operator (which syntactically is an argument to [or [[or whichever test you are using).

Section 36.2: $@

"$@" expands to all of the command line arguments as separate words. It is different from "$*", which expands to all of the arguments as a single word.

"$@" is especially useful for looping through arguments and handling arguments with spaces.

Consider we are in a script that we invoked with two arguments, like so:

```
$ ./script.sh "␣1␣2␣" "␣3␣␣4␣"
```

The variables $* or $@ will expand into $1␣$2, which in turn expand into 1␣2␣3␣4 so the loop below:

```
for var in $*; do # same for var in $@; do
    echo \\<"$var"\\>
done
```

will print for both

```
<1>
<2>
<3>
<4>
```

While "$*" will be expanded into "$1␣$2" which will in turn expand into "␣1␣2␣␣␣3␣␣4␣" and so the loop:

```
for var in "$*"; do
    echo \\<"$var"\\>
done
```

will only invoke echo once and will print

```
<␣1␣2␣␣␣3␣␣4␣>
```

And finally "$@" will expand into "$1" "$2", which will expand into "␣1␣2␣" "␣3␣␣4␣" and so the loop

```
for var in "$@"; do
    echo \\<"$var"\\>
done
```

will print

```
<␣1␣2␣>
<␣3␣␣4␣>
```

thereby preserving both the internal spacing in the arguments and the arguments separation. Note that the

construction `for var in "$@"; do ...` is so common and idiomatic that it is the default for a for loop and can be shortened to `for var; do ....`

Section 36.3: $#

To get the number of command line arguments or positional parameters - type:

```
#!/bin/bash
echo "$#"
```

When run with three arguments the example above will result with the output:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
3
```

Section 36.4: $HISTSIZE

The maximum number of remembered commands:

```
~> $ echo $HISTSIZE
1000
```

Section 36.5: $FUNCNAME

To get the name of the current function - type:

```
my_function()
{
    echo "This function is $FUNCNAME"    # This will output "This function is my_function"
}
```

This instruction will return nothing if you type it outside the function:

```
my_function

echo "This function is $FUNCNAME"    # This will output "This function is"
```

Section 36.6: $HOME

The home directory of the user

```
~> $ echo $HOME
/home/user
```

Section 36.7: $IFS

Contains the Internal Field Separator string that bash uses to split strings when looping etc. The default is the white space characters: `\n` (newline), `\t` (tab) and space. Changing this to something else allows you to split strings using different characters:

```
IFS=","
INPUTSTR="a,b,c,d"
for field in ${INPUTSTR}; do
    echo $field
```

```
done
```

The output of the above is:

```
a
b
c
d
```

Notes:

- This is responsible for the phenomenon known as word splitting.

Section 36.8: $OLDPWD

OLDPWD (**OLDP**rint**W**orking**D**irectory) contains directory before the last `cd` command:

```
~> $ cd directory
directory> $ echo $OLDPWD
/home/user
```

Section 36.9: $PWD

PWD (**P**rint**W**orking**D**irectory) The current working directory you are in at the moment:

```
~> $ echo $PWD
/home/user
~> $ cd directory
directory> $ echo $PWD
/home/user/directory
```

Section 36.10: $1 $2 $3 etc..

Positional parameters passed to the script from either the command line or a function:

```
#!/bin/bash
# $n is the n'th positional parameter
echo "$1"
echo "$2"
echo "$3"
```

The output of the above is:

```
~> $ ./testscript.sh firstarg secondarg thirdarg
firstarg
secondarg
thirdarg
```

If number of positional argument is greater than nine, curly braces must be used.

```
#  "set -- " sets positional parameters
set -- 1 2 3 4 5 6 7 8 nine ten eleven twelve
# the following line will output 10 not 1 as the value of $1 the digit 1
# will be concatenated with the following 0
echo $10   # outputs 1
echo ${10} # outputs ten
```

```
# to show this clearly:
set -- arg{1..12}
echo $10
echo ${10}
```

Section 36.11: $*

Will return all of the positional parameters in a single string.

testscript.sh:

```
#!/bin/bash
echo "$*"
```

Run the script with several arguments:

```
./testscript.sh firstarg secondarg thirdarg
```

Output:

```
firstarg secondarg thirdarg
```

Section 36.12: $!

The Process ID (pid) of the last job run in the background:

```
~> $ ls &
testfile1 testfile2
[1]+  Done                    ls
~> $ echo $!
21715
```

Section 36.13: $?

The exit status of the last executed function or command. Usually 0 will mean OK anything else will indicate a failure:

```
~> $ ls *.blah;echo $?
ls: cannot access *.blah: No such file or directory
2
~> $ ls;echo $?
testfile1 testfile2
0
```

Section 36.14: $$

The Process ID (pid) of the current process:

```
~> $ echo $$
13246
```

Section 36.15: $RANDOM

Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to

this variable seeds the random number generator (source).

```
~> $ echo $RANDOM
27119
~> $ echo $RANDOM
1349
```

Section 36.16: $BASHPID

Process ID (pid) of the current instance of Bash. This is not the same as the $$ variable, but it often gives the same result. This is new in Bash 4 and doesn't work in Bash 3.

```
~> $ echo "\$\$ pid = $$  BASHPID = $BASHPID"
$$ pid = 9265  BASHPID = 9265
```

Section 36.17: $BASH_ENV

An environment variable pointing to the Bash startup file which is read when a script is invoked.

Section 36.18: $BASH_VERSINFO

An array containing the full version information split into elements, much more convenient than `$BASH_VERSION` if you're just looking for the major version:

```
~> $ for ((i=0; i<=5; i++)); do echo "BASH_VERSINFO[$i] = ${BASH_VERSINFO[$i]}"; done
BASH_VERSINFO[0] = 3
BASH_VERSINFO[1] = 2
BASH_VERSINFO[2] = 25
BASH_VERSINFO[3] = 1
BASH_VERSINFO[4] = release
BASH_VERSINFO[5] = x86_64-redhat-linux-gnu
```

Section 36.19: $BASH_VERSION

Shows the version of bash that is running, this allows you to decide whether you can use any advanced features:

```
~> $ echo $BASH_VERSION
4.1.2(1)-release
```

Section 36.20: $EDITOR

The default editor that will be involked by any scripts or programs, usually vi or emacs.

```
~> $ echo $EDITOR
vi
```

Section 36.21: $HOSTNAME

The hostname assigned to the system during startup.

```
~> $ echo $HOSTNAME
mybox.mydomain.com
```

Section 36.22: $HOSTTYPE

This variable identifies the hardware, it can be useful in determining which binaries to execute:

```
~> $ echo $HOSTTYPE
x86_64
```

Section 36.23: $MACHTYPE

Similar to `$HOSTTYPE` above, this also includes information about the OS as well as hardware

```
~> $ echo $MACHTYPE
x86_64-redhat-linux-gnu
```

Section 36.24: $OSTYPE

Returns information about the type of OS running on the machine, eg.

```
~> $ echo $OSTYPE
linux-gnu
```

Section 36.25: $PATH

The search path for finding binaries for commands. Common examples include `/usr/bin` and `/usr/local/bin`.

When a user or script attempts to run a command, the paths in `$PATH` are searched in order to find a matching file with execute permission.

The directories in `$PATH` are separated by a `:` character.

```
~> $ echo "$PATH"
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin
```

So, for example, given the above `$PATH`, if you type `lss` at the prompt, the shell will look for `/usr/kerberos/bin/lss`, then `/usr/local/bin/lss`, then `/bin/lss`, then `/usr/bin/lss`, in this order, before concluding that there is no such command.

Section 36.26: $PPID

The Process ID (pid) of the script or shell's parent, meaning the process than invoked the current script or shell.

```
~> $ echo $$
13016
~> $ echo $PPID
13015
```

Section 36.27: $SECONDS

The number of seconds a script has been running. This can get quite large if shown in the shell:

```
~> $ echo $SECONDS
98834
```

Section 36.28: $SHELLOPTS

A readonly list of the options bash is supplied on startup to control its behaviour:

```
~> $ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

Section 36.29: $_

Outputs the last field from the last command executed, useful to get something to pass onwards to another command:

```
 ~> $ ls *.sh;echo $_
testscript1.sh  testscript2.sh
testscript2.sh
```

It gives the script path if used before any other commands:

test.sh:

```
#!/bin/bash
echo "$_"
```

Output:

```
~> $ ./test.sh # running test.sh
./test.sh
```

Note: This is not a foolproof way to get the script path

Section 36.30: $GROUPS

An array containing the numbers of groups the user is in:

```
#!/usr/bin/env bash
echo You are assigned to the following groups:
for group in ${GROUPS[@]}; do
   IFS=: read -r name dummy number members < <(getent group $group )
   printf "name: %-10s number: %-15s members: %s\n" "$name" "$number" "$members"
done
```

Section 36.31: $LINENO

Outputs the line number in the current script. Mostly useful when debugging scripts.

```
#!/bin/bash
# this is line 2
echo something  # this is line 3
echo $LINENO # Will output 4
```

Section 36.32: $SHLVL

When the bash command is executed a new shell is opened. The $SHLVL environment variable holds the number of shell levels the *current* shell is running on top of.

In a *new* terminal window, executing the following command will produce different results based on the Linux distribution in use.

```
echo $SHLVL
```

Using *Fedora 25*, the output is "3". This indicates, that when opening a new shell, an initial bash command executes and performs a task. The initial bash command executes a child process (another bash command) which, in turn, executes a final bash command to open the new shell. When the new shell opens, it is running as a child process of 2 other shell processes, hence the output of "3".

In the following example (given the user is running Fedora 25), the output of $SHLVL in a new shell will be set to "3". As each bash command is executed, $SHLVL increments by one.

```
~> $ echo $SHLVL
3
~> $ bash
~> $ echo $SHLVL
4
~> $ bash
~> $ echo $SHLVL
5
```

One can see that executing the 'bash' command (or executing a bash script) opens a new shell. In comparison, sourcing a script runs the code in the current shell.

test1.sh

```
#!/usr/bin/env bash
echo "Hello from test1.sh. My shell level is $SHLVL"
source "test2.sh"
```

test2.sh

```
#!/usr/bin/env bash
echo "Hello from test2.sh. My shell level is $SHLVL"
```

run.sh

```
#!/usr/bin/env bash
echo "Hello from run.sh. My shell level is $SHLVL"
./test1.sh
```

Execute:

```
chmod +x test1.sh && chmod +x run.sh
./run.sh
```

Output:

```
Hello from run.sh. My shell level is 4
Hello from test1.sh. My shell level is 5
Hello from test2.sh. My shell level is 5
```

Section 36.33: $UID

A read only variable that stores the users' ID number:

```
~> $ echo $UID
12345
```

Chapter 37: Job Control

Section 37.1: List background processes

```
$ jobs
[1]   Running                 sleep 500 &  (wd: ~)
[2]-  Running                 sleep 600 &  (wd: ~)
[3]+  Running                 ./Fritzing &
```

First field shows the job ids. The + and - sign that follows the job id for two jobs denote the default job and next candidate default job when the current default job ends respectively. The default job is used when the `fg` or `bg` commands are used without any argument.

Second field gives the status of the job. Third field is the command used to start the process.

The last field (wd: ~) says that the sleep commands were started from the working directory ~ (Home).

Section 37.2: Bring a background process to the foreground

```
$ fg %2
sleep 600
```

%2 specifies job no. 2. If fg is used without any arguments if brings the last process put in background to the foreground.

```
$ fg %?sle
sleep 500
```

`?sle` refers to the baground process command containing "sle". If multiple background commands contain the string, it will produce an error.

Section 37.3: Restart stopped background process

```
$ bg
[8]+ sleep 600 &
```

Section 37.4: Run command in background

```
$ sleep 500 &
[1] 7582
```

Puts the sleep command in background. 7582 is the process id of the background process.

Section 37.5: Stop a foreground process

Press Ctrl + Z to stop a foreground process and put it in background

```
$ sleep 600
^Z
[8]+  Stopped                 sleep 600
```

Chapter 38: Case statement

Section 38.1: Simple case statement

In its simplest form supported by all versions of bash, case statement executes the case that matches the pattern. `;;` operator breaks after the first match, if any.

```
#!/bin/bash

var=1
case $var in
1)
  echo "Antartica"
 ;;
2)
  echo "Brazil"
 ;;
3)
  echo "Cat"
 ;;
esac
```

Outputs:

```
Antartica
```

Section 38.2: Case statement with fall through

Version ≥ 4.0

Since bash 4.0, a new operator `;&` was introduced which provides fall through mechanism.

#!/bin/bash

```
var=1
case $var in
1)
  echo "Antartica"
  ;&
2)
  echo "Brazil"
  ;&
3)
  echo "Cat"
  ;&
esac
```

Outputs:

```
Antartica
Brazil
Cat
```

Section 38.3: Fall through only if subsequent pattern(s) match

Version ≥ 4.0

Since Bash 4.0, another operator `;;&` was introduced which also provides fall through *only if* the patterns in subsequent case statement(s), if any, match.

```
#!/bin/bash

var=abc
case $var in
a*)
  echo "Antartica"
  ;;&
xyz)
  echo "Brazil"
  ;;&
*b*)
  echo "Cat"
  ;;&
esac
```

Outputs:

```
Antartica
Cat
```

In the below example, the `abc` matches both first and third case but not the second case. So, second case is not executed.

Chapter 39: Read a file (data stream, variable) line-by-line (and/or field-by-field)?

Parameter	Details
IFS	Internal field separator
file	A file name/path
`-r`	Prevents backslash interpretation when used with read
`-t`	Removes a trailing newline from each line read by `readarray`
`-d DELIM`	Continue until the first character of DELIM is read (with `read`), rather than newline

Section 39.1: Looping through a file line by line

```
while IFS= read -r line; do
   echo "$line"
done <file
```

If file may not include a newline at the end, then:

```
while IFS= read -r line || [ -n "$line" ]; do
   echo "$line"
done <file
```

Section 39.2: Looping through the output of a command field by field

Let's assume that the field separator is `:`

```
while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done < <(ping google.com)
```

Or with a pipe:

```
ping google.com | while IFS= read -d : -r field || [ -n "$field" ];do
    echo "**$field**"
done
```

Section 39.3: Read lines of a file into an array

```
readarray -t arr <file
```

Or with a loop:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <file
```

Section 39.4: Read lines of a string into an array

```
var='line 1
line 2
line3'
readarray -t arr <<< "$var"
```

or with a loop:

```
arr=()
while IFS= read -r line; do
   arr+=("$line")
done <<< "$var"
```

Section 39.5: Looping through a string line by line

```
var='line 1
line 2
line3'
while IFS= read -r line; do
   echo "-$line-"
done <<< "$var"
```

or

```
readarray -t arr <<< "$var"
for i in "${arr[@]}";do
    echo "-$i-"
done
```

Section 39.6: Looping through the output of a command line by line

```
while IFS= read -r line;do
    echo "**$line**"
done < <(ping google.com)
```

or with a pipe:

```
ping google.com |
while IFS= read -r line;do
    echo "**$line**"
done
```

Section 39.7: Read a file field by field

Let's assume that the field separator is `:` (colon) in the file *file*.

```
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "$field"
done <file
```

For a content:

```
first : se
```

```
con
d:
    Thi rd:
    Fourth
```

The output is:

```
**first **
** se
con
d**
**
    Thi rd**
**
    Fourth
**
```

Section 39.8: Read a string field by field

Let's assume that the field separator is `:`

```
var='line: 1
line: 2
line3'
while IFS= read -d : -r field || [ -n "$field" ]; do
   echo "-$field-"
done <<< "$var"
```

Output:

```

-line-
- 1
line-
- 2
line3
-
```

Section 39.9: Read fields of a file into an array

Let's assume that the field separator is `:`

```
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <file
```

Section 39.10: Read fields of a string into an array

Let's assume that the field separator is `:`

```
var='1:2:3:4:
newline'
arr=()
while IFS= read -d : -r field || [ -n "$field" ]; do
   arr+=("$field")
done <<< "$var"
```

```
echo "${arr[4]}"
```

Output:

```
newline
```

Section 39.11: Reads file (/etc/passwd) line by line and field by field

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS=: read -r username password userid groupid comment homedir cmdshell
do
  echo "$username, $userid, $comment $homedir"
done < $FILENAME
```

In unix password file, user information is stored line by line, each line consisting of information for a user separated by colon (:) character. In this example while reading the file line by line, the line is also split into fields using colon character as delimiter which is indicated by the value given for IFS.

Sample input

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Sample Output

```
mysql, 27, MySQL Server /var/lib/mysql
pulse, 497, PulseAudio System Daemon /var/run/pulse
sshd, 74, Privilege-separated SSH /var/empty/sshd
tomcat, 91, Apache Tomcat /usr/share/tomcat6
webalizer, 67, Webalizer /var/www/usage
```

To read line by line and have the entire line assigned to variable, following is a modified version of the example. Note that we have only one variable by name line mentioned here.

```
#!/bin/bash
FILENAME="/etc/passwd"
while IFS= read -r line
do
  echo "$line"
done < $FILENAME
```

Sample Input

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Sample Output

```
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pulse:x:497:495:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tomcat:x:91:91:Apache Tomcat:/usr/share/tomcat6:/sbin/nologin
webalizer:x:67:67:Webalizer:/var/www/usage:/sbin/nologin
```

Chapter 40: File execution sequence

`.bash_profile`, `.bash_login`, `.bashrc`, and `.profile` all do pretty much the same thing: set up and define functions, variables, and the sorts.

The main difference is that `.bashrc` is called at the opening of a non-login but interactive window, and `.bash_profile` and the others are called for a login shell. Many people have their `.bash_profile` or similar call `.bashrc` anyway.

Section 40.1: .profile vs .bash_profile (and .bash_login)

`.profile` is read by most shells on startup, including bash. However, `.bash_profile` is used for configurations specific to bash. For general initialization code, put it in `.profile`. If it's specific to bash, use `.bash_profile`.

`.profile` isn't actually designed for bash specifically, `.bash_profile` is though instead. (`.profile` is for Bourne and other similar shells, which bash is based off) Bash will fall back to `.profile` if `.bash_profile` isn't found.

`.bash_login` is a fallback for `.bash_profile`, if it isn't found. Generally best to use `.bash_profile` or `.profile` instead.

Chapter 41: Splitting Files

Sometimes it's useful to split a file into multiple separate files. If you have large files, it might be a good idea to break it into smaller chunks

Section 41.1: Split a file

Running the split command without any options will split a file into 1 or more separate files containing up to 1000 lines each.

```
split file
```

This will create files named `xaa`, `xab`, `xac`, etc, each containing up to 1000 lines. As you can see, all of them are prefixed with the letter `x` by default. If the initial file was less than 1000 lines, only one such file would be created.

To change the prefix, add your desired prefix to the end of the command line

```
split file customprefix
```

Now files named `customprefixaa`, `customprefixab`, `customprefixac` etc. will be created

To specify the number of lines to output per file, use the `-l` option. The following will split a file into a maximum of 5000 lines

```
split -l5000 file
```

OR

```
split --lines=5000 file
```

Alternatively, you can specify a maximum number of bytes instead of lines. This is done by using the `-b` or `--bytes` options. For example, to allow a maximum of 1MB

```
split --bytes=1MB file
```

Chapter 42: File Transfer using scp

Section 42.1: scp transferring file

To transfer a file securely to another machine - type:

```
scp file1.txt tom@server2:$HOME
```

This example presents transferring `file1.txt` from our host to `server2`'s user `tom`'s home directory.

Section 42.2: scp transferring multiple files

`scp` can also be used to transfer multiple files from one server to another. Below is example of transferring all files from `my_folder` directory with extension `.txt` to `server2`. In Below example all files will be transferred to user `tom` home directory.

```
scp /my_folder/*.txt tom@server2:$HOME
```

Section 42.3: Downloading file using scp

To download a file from remote server to the local machine - type:

```
scp tom@server2:$HOME/file.txt /local/machine/path/
```

This example shows how to download the file named `file.txt` from user `tom`'s home directory to our local machine's current directory.

Chapter 43: Pipelines

Section 43.1: Using |&

`|&` connects standard output and standard error of the first command to the second one while `|` only connects standard output of the first command to the second command.

In this example, the page is downloaded via `curl`. with `-v` option `curl` writes some info on `stderr` including , the downloaded page is written on `stdout`. Title of page can be found between **`<title>`** and **`</title>`**.

```
curl -vs 'http://www.google.com/' |& awk '/Host:/{print}
/<title>/{match($0,/<title>(.*)<\/title>/,a);print a[1]}'
```

Output is:

```
> Host: www.google.com
Google
```

But with `|` a lot more information will be printed, i.e. those that are sent to `stderr` because only `stdout` is piped to the next command. In this example all lines except the last line (Google) were sent to `stderr` by `curl`:

```
* Hostname was NOT found in DNS cache
*   Trying 172.217.20.228...
* Connected to www.google.com (172.217.20.228) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: www.google.com
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Date: Sun, 24 Jul 2016 19:04:59 GMT
< Expires: -1
< Cache-Control: private, max-age=0
< Content-Type: text/html; charset=ISO-8859-1
< P3P: CP="This is not a P3P policy! See
https://www.google.com/support/accounts/answer/151657?hl=en for more info."
< Server: gws
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< Set-Cookie: NID=82=jX0yZLPPUE7u13kKNevUCDg8yG9Ze_C03o0IM-
EopOSKL0mMITEagIE816G55L2wrTlQwgXkhq4ApFvvYEoaWF-
oEoq2T0sBTuQVdsIFULj9b2O8X35O0sAgUnc3a3JnTRBqelMcuS9QkQA; expires=Mon, 23-Jan-2017 19:04:59 GMT;
path=/; domain=.google.com; HttpOnly
< Accept-Ranges: none
< Vary: Accept-Encoding
< X-Cache: MISS from jetsib_appliance
< X-Loop-Control: 5.202.190.157 81E4F9836653D5812995BA53992F8065
< Connection: close
<
{ [data not shown]
* Closing connection 0
Google
```

Section 43.2: Show all processes paginated

```
ps -e | less
```

`ps -e` shows all the processes, its output is connected to the input of more via `|`, `less` paginates the results.

Section 43.3: Modify continuous output of a command

```
~$ ping -c 1 google.com # unmodified output
PING google.com (16.58.209.174) 56(84) bytes of data.
64 bytes from wk-in-f100.1e100.net (16.58.209.174): icmp_seq=1 ttl=53 time=47.4 ms
~$ ping google.com | grep -o '^[0-9]\+[^()]\+' # modified output
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
64 bytes from wk-in-f100.1e100.net
...
```

The pipe (`|`) connects the `stdout` of `ping` to the `stdin` of `grep`, which processes it immediately. Some other commands like `sed` default to buffering their `stdin`, which means that it has to receive enough data, before it will print anything, potentially causing delays in further processing.

Chapter 44: Managing PATH environment variable

Parameter	Details
PATH	Path environment variable

Section 44.1: Add a path to the PATH environment variable

The PATH environment variable is generally defined in ~/.bashrc or ~/.bash_profile or /etc/profile or ~/.profile or /etc/bash.bashrc (distro specific Bash configuration file)

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:
/usr/lib/jvm/jdk1.8.0_92/bin:/usr/lib/jvm/jdk1.8.0_92/db/bin:/usr/lib/jvm/jdk1.8.0_92/jre/bin
```

Now, if we want to add a path (e.g `~/bin`) to the PATH variable:

```
PATH=~/bin:$PATH
# or
PATH=$PATH:~/bin
```

But this will modify the PATH only in the current shell (and its subshell). Once you exit the shell, this modification will be gone.

To make it permanent, we need to add that bit of code to the ~/.bashrc (or whatever) file and reload the file.

If you run the following code (in terminal), it will add `~/bin` to the PATH permanently:

```
echo 'PATH=~/bin:$PATH' >> ~/.bashrc && source ~/.bashrc
```

Explanation:

- `echo 'PATH=~/bin:$PATH' >> ~/.bashrc` adds the line `PATH=~/bin:$PATH` at the end of ~/.bashrc file (you could do it with a text editor)
- `source ~/.bashrc` reloads the ~/.bashrc file

This is a bit of code (run in terminal) that will check if a path is already included and add the path only if not:

```
path=~/bin            # path to be included
bashrc=~/.bashrc      # bash file to be written and reloaded
# run the following code unmodified
echo $PATH | grep -q "\(^\|:\)$path\(:\|/\{0,1\}$\)" || echo "PATH=\$PATH:$path" >> "$bashrc";
source "$bashrc"
```

Section 44.2: Remove a path from the PATH environment variable

To remove a PATH from a PATH environment variable, you need to edit ~/.bashrc or ~/.bash_profile or /etc/profile or ~/.profile or /etc/bash.bashrc (distro specific) file and remove the assignment for that particular path.

Instead of finding the exact assignment, you could just do a replacement in the `$PATH` in its final stage.

The following will safely remove `$path` from `$PATH`:

```
path=~/bin
PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
```

To make it permanent, you will need to add it at the end of your bash configuration file.

You can do it in a functional way:

```
rpath(){
    for path in "$@";do
        PATH="$(echo "$PATH" |sed -e "s#\(^\|:\)$(echo "$path" |sed -e 's/[^^]/[&]/g' -e
's/\^/\\^/g')\(:\|/\{0,1\}$\)#\1\2#" -e 's#:\+#:#g' -e 's#^:\|:$##g')"
    done
    echo "$PATH"
}

PATH="$(rpath ~/bin /usr/local/sbin /usr/local/bin)"
PATH="$(rpath /usr/games)"
# etc ...
```

This will make it easier to handle multiple paths.

Notes:

- You will need to add these codes in the Bash configuration file (~/.bashrc or whatever).
- Run `source ~/.bashrc` to reload the Bash configuration (~/.bashrc) file.

Chapter 45: Word splitting

Parameter	Details
IFS	Internal field separator
-x	Print commands and their arguments as they are executed (Shell option)

Section 45.1: What, when and Why?

When the shell performs *parameter expansion*, *command substitution*, *variable or arithmetic expansion*, it scans for word boundaries in the result. If any word boundary is found, then the result is split into multiple words at that position. The word boundary is defined by a shell variable `IFS` (Internal Field Separator). The default value for IFS are space, tab and newline, i.e. word splitting will occur on these three white space characters if not prevented explicitly.

```
set -x
var='I am
a
multiline string'
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

In the above example this is how the `fun` function is being executed:

```
fun I am a multiline string
```

> `$var` is split into 5 args, only `I`, `am` and `a` will be printed.

Section 45.2: Bad effects of word splitting

```
$ a='I am a string with spaces'
$ [ $a = $a ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = $a ]` was interpreted as `[ I am a string with spaces = I am a string with spaces ]`. `[` is the **`test`** command for which `I am a string with spaces` is not a single argument, rather it's **6** arguments!!

```
$ [ $a = something ] || echo "didn't match"
bash: [: too many arguments
didn't match
```

> `[ $a = something ]` was interpreted as `[ I am a string with spaces = something ]`

```
$ [ $(grep . file) = 'something' ]
```

```
bash: [: too many arguments
```

> The **grep** command returns a multiline string with spaces, so you can just imagine how many arguments are there...:D

See what, when and why for the basics.

Section 45.3: Usefulness of word splitting

There are some cases where word splitting can be useful:

Filling up array:

```
arr=($(grep -o '[0-9]\+' file))
```

> This will fill up `arr` with all numeric values found in *file*

Looping through space separated words:

```
words='foo bar baz'
for w in $words;do
    echo "W: $w"
done
```

Output:

```
W: foo
W: bar
W: baz
```

Passing space separated parameters which don't contain white spaces:

```
packs='apache2 php php-mbstring php-mysql'
sudo apt-get install $packs
```

or

```
packs='
apache2
php
php-mbstring
php-mysql
'
sudo apt-get install $packs
```

> This will install the packages. If you double quote the `$packs` then it will throw an error.

> Unquoetd `$packs` is sending all the space separated package names as arguments to **apt-get**, while quoting it will send the `$packs` string as a single argument and then **apt-get** will try to install a package named `apache2 php php-mbstring php-mysql` (for the first one) which obviously doesn't exist

See what, when and why for the basics.

Section 45.4: Splitting by separator changes

We can just do simple replacement of separators from space to new line, as following example.

```
echo $sentence | tr " " "\n"
```

It'll split the value of the variable `sentence` and show it line by line respectively.

Section 45.5: Splitting with IFS

To be more clear, let's create a script named `showarg`:

```
#!/usr/bin/env bash
printf "%d args:" $#
printf " <%s>" "$@"
echo
```

Now let's see the differences:

```
$ var="This is an example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> `$var` is split into 4 args. `IFS` is white space characters and thus word splitting occurred in spaces

```
$ var="This/is/an/example"
$ showarg $var
1 args: <This/is/an/example>
```

> In above word splitting didn't occur because the `IFS` characters weren't found.

Now let's set `IFS=/`

```
$ IFS=/
$ var="This/is/an/example"
$ showarg $var
4 args: <This> <is> <an> <example>
```

> The `$var` is splitting into 4 arguments not a single argument.

Section 45.6: IFS & word splitting

See what, when and why if you don't know about the affiliation of IFS to word splitting

let's set the IFS to space character only:

```
set -x
var='I am
```

```
a
multiline string'
IFS=' '
fun() {
    echo "-$1-"
    echo "*$2*"
    echo ".$3."
}
fun $var
```

This time word splitting will only work on spaces. The `fun` function will be executed like this:

```
fun I 'am
a
multiline' string
```

> `$var` is split into 3 args. `I`, `am\na\nmultiline` and `string` will be printed

Let's set the IFS to newline only:

```
IFS=$'\n'
...
```

Now the `fun` will be executed like:

```
fun 'I am' a 'multiline string'
```

> `$var` is split into 3 args. `I am`, `a`, `multiline string` will be printed

Let's see what happens if we set IFS to nullstring:

```
IFS=
...
```

This time the `fun` will be executed like this:

```
fun 'I am
a
multiline string'
```

> `$var` is not split i.e it remained a single arg.

You can prevent word splitting by setting the IFS to nullstring

A general way of preventing word splitting is to use double quote:

```
fun "$var"
```

will prevent word splitting in all the cases discussed above i.e the `fun` function will be executed with only one argument.

Chapter 46: Avoiding date using printf

In Bash 4.2, a shell built-in time conversion for `printf` was introduced: the format specification `%(datefmt)T` makes `printf` output the date-time string corresponding to the format string `datefmt` as understood by `strftime`.

Section 46.1: Get the current date

```
$ printf '%(%F)T\n'
2016-08-17
```

Section 46.2: Set variable to current time

```
$ printf -v now '%(%T)T'
$ echo "$now"
12:42:47
```

Chapter 47: Using "trap" to react to signals and system events

Parameter	Meaning
-p	List currently installed traps
-l	List signal names and corresponding numbers

Section 47.1: Introduction: clean up temporary files

You can use the `trap` command to "trap" signals; this is the shell equivalent of the `signal()` or `sigaction()` call in C and most other programming languages to catch signals.

One of the most common uses of `trap` is to clean up temporary files on both an expected and unexpected exit.

Unfortunately not enough shell scripts do this :-(

```
#!/bin/sh

# Make a cleanup function
cleanup() {
  rm --force -- "${tmp}"
}

# Trap the special "EXIT" group, which is always run when the shell exits.
trap cleanup EXIT

# Create a temporary file
tmp="$(mktemp -p /tmp tmpfileXXXXXXX)"

echo "Hello, world!" >> "${tmp}"

# No rm -f "$tmp" needed. The advantage of using EXIT is that it still works
# even if there was an error or if you used exit.
```

Section 47.2: Catching SIGINT or Ctl+C

The trap is reset for subshells, so the `sleep` will still act on the `SIGINT` signal sent by `^C` (usually by quitting), but the parent process (i.e. the shell script) won't.

```
#!/bin/sh

# Run a command on signal 2 (SIGINT, which is what ^C sends)
sigint() {
    echo "Killed subshell!"
}
trap sigint INT

# Or use the no-op command for no output
#trap : INT

# This will be killed on the first ^C
echo "Sleeping..."
sleep 500

echo "Sleeping..."
sleep 500
```

And a variant which still allows you to quit the main program by pressing `^C` twice in a second:

```
last=0
allow_quit() {
    [ $(date +%s) -lt $(( $last + 1 )) ] && exit
    echo "Press ^C twice in a row to quit"
    last=$(date +%s)
}
trap allow_quit INT
```

Section 47.3: Accumulate a list of trap work to run at exit

Have you ever forgotten to add a `trap` to clean up a temporary file or do other work at exit?

Have you ever set one trap which canceled another?

This code makes it easy to add things to be done on exit one item at a time, rather than having one large `trap` statement somewhere in your code, which may be easy to forget.

```
# on_exit and add_on_exit
# Usage:
#   add_on_exit rm -f /tmp/foo
#   add_on_exit echo "I am exiting"
#   tempfile=$(mktemp)
#   add_on_exit rm -f "$tempfile"
# Based on http://www.linuxjournal.com/content/use-bash-trap-statement-cleanup-temporary-files
function on_exit()
{
    for i in "${on_exit_items[@]}"
    do
        eval $i
    done
}
function add_on_exit()
{
    local n=${#on_exit_items[*]}
    on_exit_items[$n]="$*"
    if [[ $n -eq 0 ]]; then
        trap on_exit EXIT
    fi
}
```

Section 47.4: Killing Child Processes on Exit

Trap expressions don't have to be individual functions or programs, they can be more complex expressions as well.

By combining `jobs -p` and `kill`, we can kill all spawned child processes of the shell on exit:

```
trap 'jobs -p | xargs kill' EXIT
```

Section 47.5: react on change of terminals window size

There is a signal WINCH (WINdowCHange), which is fired when one resizes a terminal window.

```
declare -x rows cols

update_size(){
```

```
  rows=$(tput lines) # get actual lines of term
  cols=$(tput cols)  # get actual columns of term
  echo DEBUG terminal window has no $rows lines and is $cols characters wide
}

trap update_size WINCH
```

Chapter 48: Chain of commands and operations

There are some means to chain commands together. Simple ones like just a ; or more complex ones like logical chains which run depending on some conditions. The third one is piping commands, which effectively hands over the output data to the next command in the chain.

Section 48.1: Counting a text pattern ocurrence

Using a pipe makes the output of a command be the input of the next one.

```
ls -1 | grep -c ".conf"
```

In this case the output of the ls command is used as the input of the grep command. The result will be the number of files that include ".conf" in their name.

This can be used to contruct chains of subsequent commands as long as needed:

```
ls -1 | grep ".conf" | grep -c .
```

Section 48.2: transfer root cmd output to user file

Often one want to show the result of a command executed by root to other users. The **tee** command allows easily to write a file with user perms from a command running as root:

```
su -c ifconfig | tee ~/results-of-ifconfig.txt
```

Only **ifconfig** runs as root.

Section 48.3: logical chaining of commands with && and ||

&& chains two commands. The second one runs only if the first one exits with success. **||** chains two commands. But second one runs only if first one exits with failure.

```
[ a = b  ] && echo "yes" || echo "no"

# if you want to run more commands within a logical chain, use curly braces
# which designate a block of commands
# They do need a ; before closing bracket so bash can diffentiate from other uses
# of curly braces
[ a = b ] && { echo "let me see."
               echo "hmmm, yes, i think it is true" ; } \
          || { echo "as i am in the negation i think "
               echo "this is false. a is a not b."  ; }
# mind the use of line continuation sign \
# only needed to chain yes block with || ....
```

Section 48.4: serial chaining of commands with semicolon

A semicolon separates just two commands.

```
echo "i am first" ; echo "i am second" ; echo " i am third"
```

Section 48.5: chaining commands with |

The **|** takes the output of the left command and pipes it as input the right command. Mind, that this is done in a subshell. Hence you cannot set values of vars of the calling process within a pipe.

```
find . -type f -a -iname '*.mp3'  | \
       while read filename; do
             mute --noise "$filename"
       done
```

Chapter 49: Type of Shells

Section 49.1: Start an interactive shell

```
bash
```

Section 49.2: Detect type of shell

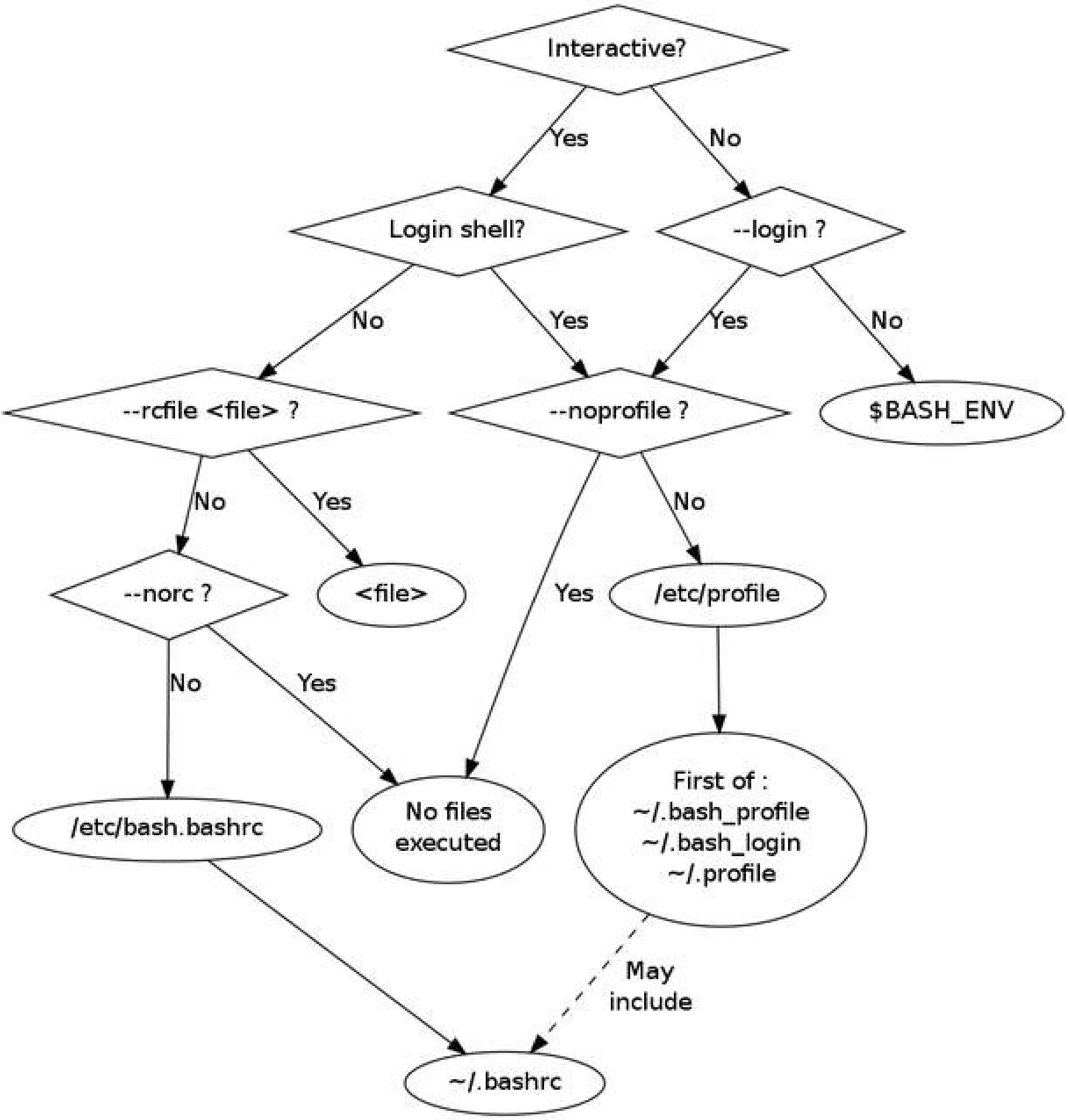
```
shopt -q login_shell && echo 'login' || echo 'not-login'
```

Section 49.3: Introduction to dot files

In Unix, files and directories beginning with a period usually contain settings for a specific program/a series of programs. Dot files are usually hidden from the user, so you would need to run `ls -a` to see them.

An example of a dot file is `.bash_history`, which contains the latest executed commands, assuming the user is using Bash.

There are various files that are sourced when you are dropped into the Bash shell. The image below, taken from this site, shows the decision process behind choosing which files to source at startup.


Interactive?
Yes
No
Login shell?
--login ?
No
Yes
Yes
No
--rcfile <file> ?
--noprofile ?
$BASH_ENV
No
Yes
No
--norc ?
<file>
Yes
/etc/profile
No
Yes
First of :
~/.bash_profile
~/.bash_login
~/.profile
/etc/bash.bashrc
No files
executed
May
include
~/.bashrc

Chapter 50: Color script output (cross-platform)

Section 50.1: color-output.sh

In the opening section of a bash script, it's possible to define some variables that function as helpers to color or otherwise format the terminal output during the run of the script.

Different platforms use different character sequences to express color. However, there's a utility called `tput` which works on all *nix systems and returns platform-specific terminal coloring strings via a consistent cross-platform API.

For example, to store the character sequence which turns the terminal text red or green:

```
red=$(tput setaf 1)
green=$(tput setaf 2)
```

Or, to store the character sequence which resets the text to default appearance:

```
reset=$(tput sgr0)
```

Then, if the BASH script needed to show different colored outputs, this can be achieved with:

cho "${green}Success!${reset}" echo "${red}Failure.${reset}"

Chapter 51: co-processes

Section 51.1: Hello World

```
# create the co-process
coproc bash

# send a command to it (echo a)
echo 'echo Hello World' >&"${COPROC[1]}"

# read a line from its output
read line <&"${COPROC[0]}"

# show the line
echo "$line"
```

The output is "Hello World".

Chapter 52: Typing variables

Section 52.1: declare weakly typed variables

declare is an internal command of bash. (internal command use **help** for displaying "manpage"). It is used to show and define variables or show function bodies.

Syntax: **declare [options] [name[=value]]...**

```
# options are used to define
# an integer
declare -i myInteger
declare -i anotherInt=10
# an array with values
declare -a anArray=( one two three)
# an assoc Array
declare -A assocArray=( [element1]="something" [second]=anotherthing )
# note that bash recognizes the string context within []

# some modifiers exist
# uppercase content
declare -u big='this will be uppercase'
# same for lower case
declare -l small='THIS WILL BE LOWERCASE'

# readonly array
declare -ra constarray=( eternal true and unchangeable )

# export integer to environment
declare -xi importantInt=42
```

You can use also the + which takes away the given attribute. Mostly useless, just for completness.

To display variables and/or functions there are some options too

```
# printing definded vars and functions
declare -f
# restrict output to functions only
declare -F # if debugging prints line number and filename defined in too
```

Chapter 53: Jobs at specific times

Section 53.1: Execute job once at specific time

Note: **at** is not installed by default on most of modern distributions.

To execute a job once at some other time than now, in this example 5pm, you can use

```
echo "somecommand &" | at 5pm
```

If you want to catch the output, you can do that in the usual way:

```
echo "somecommand > out.txt 2>err.txt &" | at 5pm
```

`at` understands many time formats, so you can also say

```
echo "somecommand &" | at now + 2 minutes
echo "somecommand &" | at 17:00
echo "somecommand &" | at 17:00 Jul 7
echo "somecommand &" | at 4pm 12.03.17
```

If no year or date are given, it assumes the next time the time you specified occurs. So if you give a hour that already passed today, it will assume tomorrow, and if you give a month that already passed this year, it will assume next year.

This also works together with nohup like you would expect.

```
echo "nohup somecommand > out.txt 2>err.txt &" | at 5pm
```

There are some more commands to control timed jobs:

- **atq** lists all timed jobs (**atq**ueue)
- **atrm** removes a timed job (**atre**m**o**ve)
- **batch** does basically the same like at, but runs jobs only when system load is lower than 0.8

All commands apply to jobs of the user logged in. If logged in as root, system wide jobs are handled of course.

Section 53.2: Doing jobs at specified times repeatedly using systemd.timer

systemd provides a modern implementation of **cron**. To execute a script periodical a service and a timer file ist needed. The service and timer files should be placed in /etc/systemd/{system,user}. The service file:

```
[Unit]
Description=my script or programm does the very best and this is the description

[Service]
# type is important!
Type=simple
# program|script to call. Always use absolute pathes
# and redirect STDIN and STDERR as there is no terminal while being executed
ExecStart=/absolute/path/to/someCommand >>/path/to/output 2>/path/to/STDERRoutput
#NO install section!!!! Is handled by the timer facitlities itself.
#[Install]
```

```
#WantedBy=multi-user.target
```

Next the timer file:

```
[Unit]
Description=my very first systemd timer
[Timer]
# Syntax for date/time specifications is  Y-m-d H:M:S
# a * means "each", and a comma separated list of items can be given too
# *-*-* *,15,30,45:00  says every year, every month, every day, each hour,
# at minute 15,30,45 and zero seconds

OnCalendar=*-*-* *:01:00
# this one runs each hour at one minute zero second e.g. 13:01:00
```

Chapter 54: Handling the system prompt

Escape	Details
\a	A bell character.
\d	The date, in "Weekday Month Date" format (e.g., "Tue May 26").
\D{FORMAT}	The FORMAT is passed to `strftime'(3) and the result is inserted into the prompt string; an empty FORMAT results in a locale-specific time representation. The braces are required.
\e	An escape character. \033 works of course too.
\h	The hostname, up to the first `.'. (i.e. no domain part)
\H	The hostname eventually with domain part
\j	The number of jobs currently managed by the shell.
\l	The basename of the shell's terminal device name.
\n	A newline.
\r	A carriage return.
\s	The name of the shell, the basename of `$0' (the portion following the final slash).
\t	The time, in 24-hour HH:MM:SS format.
\T	The time, in 12-hour HH:MM:SS format.
@	The time, in 12-hour am/pm format.
\A	The time, in 24-hour HH:MM format.
\u	The username of the current user.
\v	The version of Bash (e.g., 2.00)
\V	The release of Bash, version + patchlevel (e.g., 2.00.0)
\w	The current working directory, with $HOME abbreviated with a tilde (uses the $PROMPT_DIRTRIM variable).
\W	The basename of $PWD, with $HOME abbreviated with a tilde.
!	The history number of this command.
#	The command number of this command.
$	If the effective uid is 0, **#**, otherwise **$**.
\NNN	The character whose ASCII code is the octal value NNN.
\	A backslash.
\[	Begin a sequence of non-printing characters. This could be used to embed a terminal control sequence into the prompt.
\]	End a sequence of non-printing characters.

Section 54.1: Using the PROMPT_COMMAND envrionment variable

When the last command in an interactive bash instance is done, the evaluated PS1 variable is displayes. Before actually displaying PS1 bash looks whether the PROMPT_COMMAND is set. This value of this var must be a callable program or script. If this var is set this program/script is called BEFORE the PS1 prompt is displayed.

```
# just a stupid function, we will use to demonstrate
# we check the date if Hour is 12 and Minute is lower than 59
lunchbreak(){
   if (( $(date +%H) == 12 && $(date +%M) < 59 )); then
      # and print colored \033[ starts the escape sequence
      # 5; is blinking attribute
      # 2; means bold
      # 31 says red
```

```
        printf "\033[5;1;31mmind the lunch break\033[0m\n";
    else
        printf "\033[33mstill working...\033[0m\n";
    fi;
}

# activating it
export PROMPT_COMMAND=lunchbreak
```

Section 54.2: Using PS2

PS2 is displayed when a command extends to more than one line and bash awaits more keystrokes. It is displayed too when a compound command like **while...do..done** and alike is entered.

```
export PS2="would you please complete this command?\n"
# now enter a command extending to at least two lines to see PS2
```

Section 54.3: Using PS3

When the select statement is executed, it displays the given items prefixed with a number and then displays the PS3 prompt:

```
export PS3="  To choose your language type the preceding number : "
select lang in EN CA FR DE; do
    # check input here until valid.
    break
done
```

Section 54.4: Using PS4

PS4 is displayes when bash is in debugging mode.

```
#!/usr/bin/env bash

# switch on debugging
set -x

# define a stupid_func
stupid_func(){
    echo I am line 1 of stupid_func
    echo I am line 2 of stupid_func
}

# setting the PS4 "DEBUG" prompt
export PS4='\nDEBUG level:$SHLVL subshell-level: $BASH_SUBSHELL \nsource-file:${BASH_SOURCE}
line#:${LINENO} function:${FUNCNAME[0]}:+${FUNCNAME[0]}(): }\nstatement: '

# a normal statement
echo something

# function call
stupid_func

# a pipeline of commands running in a subshell
( ls -l | grep 'x' )
```

Section 54.5: Using PS1

PS1 is the normal system prompt indicating that bash waits for commands being typed in. It understands some escape sequences and can execute functions or progams. As bash has to position the cursor after the displayes prompt, it needs to know how to calculate the effective length of the prompt string. To indicate non printing sequences of chars within the PS1 variable escaped braces are used: **\[** *a non printing sequence of chars* **\]**. All being said holds true for all PS* vars.

(The black caret indicates cursor)

```
#everything not being an escape sequence will be literally printed
export PS1="literal sequence "  # Prompt is now:
literal sequence ▉

# \u == user \h == host \w == actual working directory
# mind the single quotes avoiding interpretation by shell
export PS1='\u@\h:\w > ' # \u == user, \h == host, \w actual working dir
looser@host:/some/path > ▉

# executing some commands within PS1
# following line will set foreground color to red, if user==root,
# else it resets attributes to default
# $( (($EUID == 0)) &&  tput setaf 1)
# later we do reset attributes to default with
# $(  tput sgr0 )
# assuming being root:
PS1="\[$( (($EUID == 0)) &&  tput setaf 1 \]\u\[$(tput sgr0)\]@\w:\w \$ "
looser@host:/some/path > ▉  # if not root else <red>root<default>@host....
```

Chapter 55: The cut command

Parameter	Details
-f, --fields	Field-based selection
-d, --delimiter	Delimiter for field-based selection
-c, --characters	Character-based selection, delimiter ignored or error
-s, --only-delimited	Suppress lines with no delimiter characters (printed as-is otherwise)
--complement	Inverted selection (extract all *except* specified fields/characters
--output-delimiter	Specify when it has to be different from the input delimiter

The `cut` command is a fast way to extract parts of lines of text files. It belongs to the oldest Unix commands. Its most popular implementations are the GNU version found on Linux and the FreeBSD version found on MacOS, but each flavor of Unix has its own. See below for differences. The input lines are read either from `stdin` or from files listed as arguments on the command line.

Section 55.1: Only one delimiter character

You cannot have more than one delimiter: if you specify something like `-d ",;:"`, some implementations will use only the first character as a delimiter (in this case, the comma.) Other implementations (e.g. GNU `cut`) will give you an error message.

```
$ cut -d ",;:" -f2 <<<"J.Smith,1 Main Road,cell:1234567890;land:4081234567"
cut: the delimiter must be a single character
Try `cut --help' for more information.
```

Section 55.2: Repeated delimiters are interpreted as empty fields

```
$ cut -d, -f1,3 <<<"a,,b,c,d,e"
a,b
```

is rather obvious, but with space-delimited strings it might be less obvious to some

```
$ cut -d ' ' -f1,3 <<<"a  b c d e"
a b
```

`cut` cannot be used to parse arguments as the shell and other programs do.

Section 55.3: No quoting

There is no way to protect the delimiter. Spreadsheets and similar CSV-handling software usually can recognize a text-quoting character which makes it possible to define strings containing a delimiter. With `cut` you cannot.

```
$ cut -d, -f3 <<<'John,Smith,"1, Main Street"'
"1
```

Section 55.4: Extracting, not manipulating

You can only extract portions of lines, not reorder or repeat fields.

```
$ cut -d, -f2,1 <<<'John,Smith,USA' ## Just like -f1,2
```

```
John,Smith
$ cut -d, -f2,2 <<<'John,Smith,USA' ## Just like -f2
Smith
```

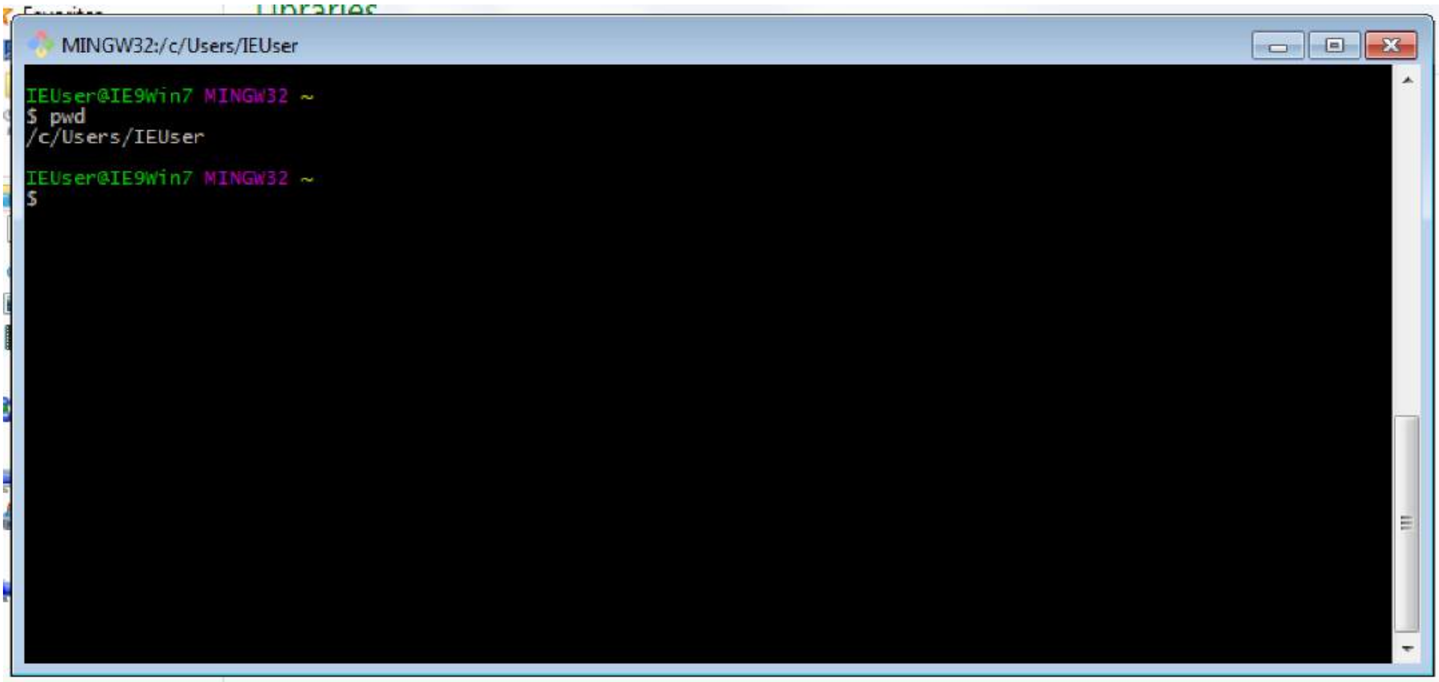
Chapter 56: Bash on Windows 10

Section 56.1: Readme

The simpler way to use Bash in Windows is to install Git for Windows. It's shipped with Git Bash which is a real Bash. You can access it with shortcut in :

`Start > All Programs > Git > Git Bash`

Commands like `grep`, `ls`, `find`, `sed`, `vi` etc is working.



Chapter 57: Cut Command

Option	Description
`-b LIST, --bytes=LIST`	Print the bytes listed in the LIST parameter
`-c LIST, --characters=LIST`	Print characters in positions specified in LIST parameter
`-f LIST, --fields=LIST`	Print fields or columns
`-d DELIMITER`	Used to separate columns or fields

In Bash, the `cut` command is useful for dividing a file into several smaller parts.

Section 57.1: Show the first column of a file

Suppose you have a file that looks like this

```
John Smith 31
Robert Jones 27
...
```

This file has 3 columns separated by spaces. To select only the first column, do the following.

```
cut -d ' ' -f1 filename
```

Here the `-d` flag, specifies the delimiter, or what separates the records. The `-f` flag specifies the field or column number. This will display the following output

```
John
Robert
...
```

Section 57.2: Show columns x to y of a file

Sometimes, it's useful to display a range of columns in a file. Suppose you have this file

```
Apple California 2017 1.00 47
Mango Oregon 2015 2.30 33
```

To select the first 3 columns do

```
cut -d ' ' -f1-3 filename
```

This will display the following output

```
Apple California 2017
Mango Oregon 2015
```

Chapter 58: global and local variables

By default, every variable in bash is **global** to every function, script and even the outside shell if you are declaring your variables inside a script.

If you want your variable to be local to a function, you can use `local` to have that variable a new variable that is independent to the global scope and whose value will only be accessible inside that function.

Section 58.1: Global variables

```
var="hello"

function foo(){
    echo $var
}

foo
```

Will obviously output "hello", but this works the other way around too:

```
function foo()  {
    var="hello"
}

foo
echo $var
```

Will also output `"hello"`

Section 58.2: Local variables

```
function foo() {
    local var
    var="hello"
}

foo
echo $var
```

Will output nothing, as var is a variable local to the function foo, and its value is not visible from outside of it.

Section 58.3: Mixing the two together

```
var="hello"

function foo(){
    local var="sup?"
    echo "inside function, var=$var"
}

foo
echo "outside function, var=$var"
```

Will output

```
inside function, var=sup?
outside function, var=hello
```

Chapter 59: CGI Scripts

Section 59.1: Request Method: GET

It is quite easy to call a CGI-Script via `GET`.
First you will need the `encoded url` of the script.

Then you add a question mark `?` followed by variables.

- Every variable should have two sections separated by **=**.
  First section should be always a unique name for each variable,
  while the second part has values in it only
- Variables are separated by **&**
- Total length of the string should not rise above **255** characters
- Names and values needs to be html-encoded (replace: **</ , / ? : @ & = + $**)
  Hint:
  When using **html-forms** the request method can be generated by it self.
  With **Ajax** you can encode all via **encodeURI** and **encodeURIComponent**

Example:

```
http://www.example.com/cgi-bin/script.sh?var1=Hello%20World!&var2=This%20is%20a%20Test.&
```

The server should communicate via **Cross-Origin Resource Sharing** (CORS) only, to make request more secure. In this showcase we use **CORS** to determine the `Data-Type` we want to use.

There are many `Data-Types` we can choose from, the most common are...

- **text/html**
- **text/plain**
- **application/json**

When sending a request, the server will also create many environment variables. For now the most important environment variables are `$REQUEST_METHOD` and `$QUERY_STRING`.

The **Request Method** has to be `GET` nothing else!
The **Query String** includes all the `html-endoded data`.

The Script

```
#!/bin/bash

# CORS is the way to communicate, so lets response to the server first
echo "Content-type: text/html"    # set the data-type we want to use
echo ""    # we don't need more rules, the empty line initiate this.

# CORS are set in stone and any communication from now on will be like reading a html-document.
# Therefor we need to create any stdout in html format!

# create html scructure and send it to stdout
echo "<!DOCTYPE html>"
echo "<html><head>"

# The content will be created depending on the Request Method
if [ "$REQUEST_METHOD" = "GET" ]; then
```

```
    # Note that the environment variables $REQUEST_METHOD and $QUERY_STRING can be processed by the
shell directly.
    # One must filter the input to avoid cross site scripting.

    Var1=$(echo "$QUERY_STRING" | sed -n 's/^.*var1=\([^&]*\).*$/\1/p')    # read value of "var1"
    Var1_Dec=$(echo -e $(echo "$Var1" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))    # html decode

    Var2=$(echo "$QUERY_STRING" | sed -n 's/^.*var2=\([^&]*\).*$/\1/p')
    Var2_Dec=$(echo -e $(echo "$Var2" | sed 's/+/ /g;s/%\(..\)/\\x\1/g;'))

    # create content for stdout
    echo "<title>Bash-CGI Example 1</title>"
    echo "</head><body>"
    echo "<h1>Bash-CGI Example 1</h1>"
    echo "<p>QUERY_STRING: ${QUERY_STRING}<br>var1=${Var1_Dec}<br>var2=${Var2_Dec}</p>"    # print
the values to stdout

else

    echo "<title>456 Wrong Request Method</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Requesting data went wrong.<br>The Request method has to be \"GET\" only!</p>"

fi

echo "<hr>"
echo "$SERVER_SIGNATURE"    # an other environment variable
echo "</body></html>"    # close html

exit 0
```

The **html-document** will look like this ...

```
<html><head>
<title>Bash-CGI Example 1</title>
</head><body>
<h1>Bash-CGI Example 1</h1>
<p>QUERY_STRING: var1=Hello%20World!&amp;var2=This%20is%20a%20Test.&amp;<br>var1=Hello
World!<br>var2=This is a Test.</p>
<hr>
<address>Apache/2.4.10 (Debian) Server at example.com Port 80</address>


</body></html>
```

The **output** of the variables will look like this ...

```
var1=Hello%20World!&var2=This%20is%20a%20Test.&
Hello World!
This is a Test.
Apache/2.4.10 (Debian) Server at example.com Port 80
```

Negative side effects...

- All the encoding and decoding does not look nice, but is needed
- The Request will be public readable and leave a tray behind
- The size of a request is limited

- Needs protection against Cross-Side-Scripting (XSS)

Section 59.2: Request Method: POST /w JSON

Using Request Method `POST` in combination with `SSL` makes datatransfer more secure.

In addition...

- Most of the encoding and decoding is not needed any more
- The URL will be visible to any one and needs to be url encoded.
  The data will be send separately and therefor should be secured via SSL
- The size of the data is almost unlitmited
- Still needs protection against Cross-Side-Scripting (XSS)

To keep this showcase simple we want to receive **JSON Data**
and communication should be over **Cross-Origin Resource Sharing** (CORS).

The following script will also demonstrate two different **Content-Types**.

```
#!/bin/bash

exec 2>/dev/null    # We don't want any error messages be printed to stdout
trap "response_with_html && exit 0" ERR    # response with an html message when an error occurred
and close the script

function response_with_html(){
    echo "Content-type: text/html"
    echo ""
    echo "<!DOCTYPE html>"
    echo "<html><head>"
    echo "<title>456</title>"
    echo "</head><body>"
    echo "<h1>456</h1>"
    echo "<p>Attempt to communicate with the server went wrong.</p>"
    echo "<hr>"
    echo "$SERVER_SIGNATURE"
    echo "</body></html>"
}

function response_with_json(){
    echo "Content-type: application/json"
    echo ""
    echo "{\"message\": \"Hello World!\"}"
}

if [ "$REQUEST_METHOD" = "POST" ]; then

    # The environment variabe $CONTENT_TYPE describes the data-type received
    case "$CONTENT_TYPE" in
    application/json)
        # The environment variabe $CONTENT_LENGTH describes the size of the data
        read -n "$CONTENT_LENGTH" QUERY_STRING_POST        # read datastream

        # The following lines will prevent XSS and check for valide JSON-Data.
        # But these Symbols need to be encoded somehow before sending to this script
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed "s/'//g" | sed
's/\$//g;s/`//g;s/\*//g;s/\\//g' )        # removes some symbols (like \ * ` $ ') to prevent XSS
with Bash and SQL.
```

```
        QUERY_STRING_POST=$(echo "$QUERY_STRING_POST" | sed -e :a -e 's/<[^>]*>//g;/</N;//ba')    #
removes most html declarations to prevent XSS within documents
        JSON=$(echo "$QUERY_STRING_POST" | jq .)        # json encode - This is a pretty save way
to check for valide json code
    ;;
    *)
        response_with_html
        exit 0
    ;;
    esac

else
    response_with_html
    exit 0
fi

# Some Commands ...

response_with_json

exit 0
```

You will get `{"message":"Hello World!"}` as an answer when sending **JSON-Data** via `POST` to this Script. Every thing else will receive the html document.

Important is also the varialbe `$JSON`. This variable is free of XSS, but still could have wrong values in it and needs to be verify first. Please keep that in mind.

This code works similar without JSON.
You could get any data this way.
You just need to change the `Content-Type` for your needs.

Example:

```
if [ "$REQUEST_METHOD" = "POST" ]; then
    case "$CONTENT_TYPE" in
    application/x-www-form-urlencoded)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    text/plain)
            read -n "$CONTENT_LENGTH" QUERY_STRING_POST
    ;;
    esac
fi
```

Last but not least, don't forget to response to all requests, otherwise third party programms won't know if they succeeded

Chapter 60: Select keyword

Select keyword can be used for getting input argument in a menu format.

Section 60.1: Select keyword can be used for getting input argument in a menu format

Suppose you want the `user` to **SELECT** keywords from a menu, we can create a script similar to

```
#!/usr/bin/env bash

select os in "linux" "windows" "mac"
do
    echo "${os}"
    break
done
```

Explanation: Here **SELECT** keyword is used to loop through a list of items that will be presented at the command prompt for a user to pick from. Notice the **break** keyword for breaking out of the loop once the user makes a choice. Otherwise, the loop will be endless!

Results: Upon running this script, a menu of these items will be displayed and the user will be prompted for a selection. Upon selection, the value will be displayed, returning back to command prompt.

```
>bash select_menu.sh
1) linux
2) windows
3) mac
#? 3
mac
>
```

Chapter 61: When to use eval

First and foremost: know what you're doing! Secondly, while you should avoid using **eval**, if its use makes for cleaner code, go ahead.

Section 61.1: Using Eval

For example, consider the following that sets the contents of `$@` to the contents of a given variable:

```
a=(1 2 3)
eval set -- "${a[@]}"
```

This code is often accompanied by **getopt** or **getopts** to set `$@` to the output of the aforementioned option parsers, however, you can also use it to create a simple `pop` function that can operate on variables silently and directly without having to store the result to the original variable:

```
isnum()
{
    # is argument an integer?
    local re='^[0-9]+$'
    if [[ -n $1 ]]; then
        [[ $1 =~ $re ]] && return 0
        return 1
    else
        return 2
    fi
}

isvar()
{
    if isnum "$1"; then
        return 1
    fi
    local arr="$(eval eval -- echo -n "\$$1")"
    if [[ -n ${arr[@]} ]]; then
        return 0
    fi
    return 1
}

pop()
{
    if [[ -z $@ ]]; then
        return 1
    fi

    local var=
    local isvar=0
    local arr=()

    if isvar "$1"; then # let's check to see if this is a variable or just a bare array
        var="$1"
        isvar=1
        arr=($(eval eval -- echo -n "\${$1[@]}")) # if it is a var, get its contents
    else
        arr=($@)
    fi
```

```
    # we need to reverse the contents of $@ so that we can shift
    # the last element into nothingness
    arr=($(awk <<<"${arr[@]}" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # set $@ to ${arr[@]} so that we can run shift against it.
    eval set -- "${arr[@]}"

    shift # remove the last element

    # put the array back to its original order
    arr=($(awk <<<"$@" '{ for (i=NF; i>1; --i) printf("%s ",$i); print $1; }'

    # echo the contents for the benefit of users and for bare arrays
    echo "${arr[@]}"

    if ((isvar)); then
        # set the contents of the original var to the new modified array
        eval -- "$var=(${arr[@]})"
    fi
}
```

Section 61.2: Using Eval with Getopt

While eval may not be needed for a `pop` like function, it is however required whenever you use `getopt`:

Consider the following function that accepts `-h` as an option:

```
f()
{
    local __me__="${FUNCNAME[0]}"
    local argv="$(getopt -o 'h' -n $__me__ -- "$@")"

    eval set -- "$argv"

    while :; do
        case "$1" in
            -h)
              echo "LOLOLOLOL"
              return 0
              ;;
            --)
              shift
              break
              ;;
    done

    echo "$@"
}
```

Without `eval set -- "$argv"` generates

`-h --`

instead of the desired `(-h --)` and subsequently enters an infinite loop because

`-h --`

doesn't match `--` or `-h`.

Chapter 62: Networking With Bash

Bash is often commonly used in the management and maintenance of servers and clusters. Information pertaining to typical commands used by network operations, when to use which command for which purpose, and examples/samples of unique and/or interesting applications of it should be included

Section 62.1: Networking commands

```
ifconfig
```

The above command will show all active interface of the machine and also give the information of

1. IP address assign to interface
2. MAC address of the interface
3. Broadcast address
4. Transmit and Receive bytes

Some example

```
ifconfig -a
```

The above command also show the disable interface

```
ifconfig eth0
```

The above command will only show the eth0 interface

```
ifconfig eth0 192.168.1.100 netmask 255.255.255.0
```

The above command will assign the static IP to eth0 interface

```
ifup eth0
```

The above command will enable the eth0 interface

```
ifdown eth0
```

The below command will disable the eth0 interface

```
ping
```

The above command (Packet Internet Grouper) is to test the connectivity between the two nodes

```
ping -c2 8.8.8.8
```

The above command will ping or test the connectivity with google server for 2 seconds.

```
traceroute
```

The above command is to use in troubleshooting to find out the number of hops taken to reach the destination.

```
netstat
```

The above command (Network statistics) give the connection info and their state

```
dig www.google.com
```

The above command (domain information grouper) query the DNS related information

```
nslookup www.google.com
```

The above command query the DNS and find out the IP address of corresponding the website name.

```
route
```

The above command is used to check the Netwrok route information. It basically show you the routing table

```
router add default gw 192.168.1.1 eth0
```

The above command will add the default route of network of eth0 Interface to 192.168.1.1 in routing table.

```
route del default
```

The above command will delete the default route from the routing table

Chapter 63: Parallel

Option	Description
`-j n`	Run n jobs in parallel
`-k`	Keep same order
`-X`	Multiple arguments with context replace
`--colsep regexp`	Split input on regexp for positional replacements
`{} {.} {/} {/.} {#}`	Replacement strings
`{3} {3.} {3/} {3/.}`	Positional replacement strings
`-S sshlogin`	`Example: foo@server.example.com`
`--trc {}.bar`	Shorthand for --transfer --return {}.bar --cleanup
`--onall`	Run the given command with argument on all sshlogins
`--nonall`	Run the given command with no arguments on all sshlogins
`--pipe`	Split stdin (standard input) to multiple jobs.
`--recend str`	Record end separator for --pipe.
`--recstart str`	Record start separator for --pipe.

Jobs in GNU Linux can be parallelized using GNU parallel. A job can be a single command or a small script that has to be run for each of the lines in the input. The typical input is a list of files, a list of hosts, a list of users, a list of URLs, or a list of tables. A job can also be a command that reads from a pipe.

Section 63.1: Parallelize repetitive tasks on list of files

Many repetitive jobs can be performed more efficiently if you utilize more of your computer's resources (i.e. CPU's and RAM). Below is an example of running multiple jobs in parallel.

Suppose you have a `< list of files >`, say output from `ls`. Also, let these files are bz2 compressed and the following order of tasks need to be operated on them.

1. Decompress the bz2 files using `bzcat` to stdout
2. Grep (e.g. filter) lines with specific keyword(s) using `grep <some key word>`
3. Pipe the output to be concatenated into one single gzipped file using `gzip`

Running this using a while-loop may look like this

```
filenames="file_list.txt"
while read -r line
do
name="$line"
     ## grab lines with puppies in them
     bzcat $line | grep puppies | gzip >> output.gz
done < "$filenames"
```

Using GNU Parallel, we can run 3 parallel jobs at once by simply doing

```
parallel -j 3 "bzcat {} | grep puppies" ::: $( cat filelist.txt ) | gzip > output.gz
```

This command is simple, concise and more efficient when number of files and file size is large. The jobs gets initiated by `parallel`, option `-j 3` launches 3 parallel jobs and input to the parallel jobs is taken in by `:::`. The output is eventually piped to `gzip > output.gz`

Section 63.2: Parallelize STDIN

Now, let's imagine we have 1 large file (e.g. 30 GB) that needs to be converted, line by line. Say we have a script, `convert.sh`, that does this **`<task>`**. We can pipe contents of this file to stdin for parallel to take in and work with in *chunks* such as

```
<stdin> | parallel --pipe --block <block size> -k <task> > output.txt
```

where **`<stdin>`** can originate from anything such as `cat <file>`.

As a reproducible example, our task will be `nl -n rz`. Take any file, mine will be `data.bz2`, and pass it to **`<stdin>`**

```
bzcat data.bz2 | nl | parallel --pipe --block 10M -k nl -n rz | gzip > ouptput.gz
```

The above example takes **`<stdin>`** from `bzcat data.bz2 | nl`, where I included `nl` just as a proof of concept that the final output `output.gz` will be saved in the order it was received. Then, `parallel` divides the **`<stdin>`** into chunks of size 10 MB, and for each chunk it passes it through `nl -n rz` where it just appends a numbers rightly justified (see `nl --help` for further details). The options `--pipe` tells `parallel` to split **`<stdin>`** into multiple jobs and `-- block` specifies the size of the blocks. The option `-k` specifies that ordering must be maintained.

Your final output should look something like

```
000001       1  <data>
000002       2  <data>
000003       3  <data>
000004       4  <data>
000005       5  <data>
 ...
000587  552409  <data>
000588  552410  <data>
000589  552411  <data>
000590  552412  <data>
000591  552413  <data>
```

My original file had 552,413 lines. The first column represents the parallel jobs, and the second column represents the original line numbering that was passed to `parallel` in chunks. You should notice that the order in the second column (and rest of the file) is maintained.

Chapter 64: Decoding URL

Section 64.1: Simple example

Encoded URL

> http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty

Use this command to decode the URL

```
echo "http%3A%2F%2Fwww.foo.com%2Findex.php%3Fid%3Dqwerty" | sed -e "s/%\([0-9A-F][0-9A-
F]\)/\\\\\x\1/g" | xargs -0 echo -e
```

Decoded URL (result of command)

> http://www.foo.com/index.php?id=qwerty

Section 64.2: Using printf to decode a string

```
#!bin/bash

$ string='Question%20-
%20%22how%20do%20I%20decode%20a%20percent%20encoded%20string%3F%22%0AAnswer%20%20%20-
%20Use%20printf%20%3A)'
$ printf '%b\n' "${string//%/\\x}"

# the result
Question - "how do I decode a percent encoded string?"
Answer   - Use printf :)
```

Chapter 65: Design Patterns

Accomplish some common design patterns in Bash

Section 65.1: The Publish/Subscribe (Pub/Sub) Pattern

When a Bash project turns into a library, it can become difficult to add new functionality. Function names, variables and parameters usually need to be changed in the scripts that utilize them. In scenarios like this, it is helpful to decouple the code and use an event driven design pattern. In said pattern, an external script can subscribe to an event. When that event is triggered (published) the script can execute the code that it registered with the event.

pubsub.sh:

```
#!/usr/bin/env bash

#
# Save the path to this script's directory in a global env variable
#
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

#
# Array that will contain all registered events
#
EVENTS=()

function action1() {
    echo "Action #1 was performed ${2}"
}

function action2() {
    echo "Action #2 was performed"
}

#
# @desc   :: Registers an event
# @param  :: string $1 - The name of the event. Basically an alias for a function name
# @param  :: string $2 - The name of the function to be called
# @param  :: string $3 - Full path to script that includes the function being called
#
function subscribe() {
    EVENTS+=("${1};${2};${3}")
}

#
# @desc   :: Public an event
# @param  :: string $1 - The name of the event being published
#
function publish() {
    for event in ${EVENTS[@]}; do
        local IFS=";"
        read -r -a event <<< "$event"
        if [[  "${event[0]}" ==  "${1}" ]]; then
            ${event[1]} "$@"
        fi
    done
}

#
# Register our events and the functions that handle them
```

```
    #
    subscribe "/do/work"           "action1" "${DIR}"
    subscribe "/do/more/work"      "action2" "${DIR}"
    subscribe "/do/even/more/work" "action1" "${DIR}"

    #
    # Execute our events
    #
    publish "/do/work"
    publish "/do/more/work"
    publish "/do/even/more/work" "again"
```

Run:

```
chmod +x pubsub.sh
./pubsub.sh
```

Chapter 66: Pitfalls

Section 66.1: Whitespace When Assigning Variables

Whitespace matters when assigning variables.

```
foo = 'bar' # incorrect
foo= 'bar'  # incorrect
foo='bar'   # correct
```

The first two will result in syntax errors (or worse, executing an incorrect command). The last example will correctly set the variable `$foo` to the text "bar".

Section 66.2: Failed commands do not stop script execution

In most scripting languages, if a function call fails, it may throw an exception and stop execution of the program. Bash commands do not have exceptions, but they do have exit codes. A non-zero exit code signals failure, however, a non-zero exit code will not stop execution of the program.

This can lead to dangerous (although admittedly contrived) situations like so:

```
#!/bin/bash
cd ~/non/existent/directory
rm -rf *
```

If `cd`-ing to this directory fails, Bash will ignore the failure and move onto the next command, wiping clean the directory from where you ran the script.

The best way to deal with this problem is to make use of the set command:

```
#!/bin/bash
set -e
cd ~/non/existent/directory
rm -rf *
```

`set -e` tells Bash to exit the script immediately if any command returns a non-zero status.

Section 66.3: Missing The Last Line in a File

The C standard says that files should end with a new line, so if EOF comes at the end of a line, that line may not be missed by some commands. As an example:

```
$ echo 'one\ntwo\nthree\c' > file.txt

$ cat file.txt
one
two
three

$ while read line ; do echo "line $line" ; done < file.txt
one
two
```

To make sure this works correctly for in the above example, add a test so that it will continue the loop if the last line is not empty.

```
$ while read line || [ -n "$line" ] ; do echo "line $line" ; done < file.txt
one
two
three
```

Appendix A: Keyboard shortcuts

Section A.1: Editing Shortcuts

Shortcut	Description
Ctrl + a	move to the beginning of the line
Ctrl + e	move to the end of the line
Ctrl + k	Kill the text from the current cursor position to the end of the line.
Ctrl + u	Kill the text from the current cursor position to the beginning of the line
Ctrl + w	Kill the word behind the current cursor position
Alt + b	move backward one word
Alt + f	move forward one word
Ctrl + Alt + e	shell expand line
Ctrl + y	Yank the most recently killed text back into the buffer at the cursor.
Alt + y	Rotate through killed text. You can only do this if the prior command is Ctrl + y or Alt + y.

Killing text will delete text, but save it so that the user can reinsert it by yanking. Similar to cut and paste except that the text is placed on a kill ring which allows for storing more than one set of text to be yanked back on to the command line.

You can find out more in the emacs manual.

Section A.2: Recall Shortcuts

Shortcut	Description
Ctrl + r	search the history backwards
Ctrl + p	previous command in history
Ctrl + n	next command in history
Ctrl + g	quit history searching mode
Alt + .	use the last word of the previous command repeat to get the last word of the previous + 1 command
Alt + n Alt + .	use the nth word of the previous command
!! + Return	execute the last command again (useful when you forgot sudo: `sudo !!`)

Section A.3: Macros

Shortcut	Description
Ctrl + x, (	start recording a macro
Ctrl + x,)	stop recording a macro
Ctrl + x, e	execute the last recorded macro

Section A.4: Custome Key Bindings

With the **`bind`** command it is possible to define custom key bindings.

The next example bind an Alt + w to `>/dev/null 2>&1`:

```
bind '"\ew"':"\" >/dev/null 2>&1\""
```

If you want to execute the line immediately add `\C-m` (Enter) to it:

```
bind '"\ew"':"\" >/dev/null 2>&1\C-m\""
```

Section A.5: Job Control

Shortcut	Description
Ctrl + c	Stop the current job
Ctrl + z	Suspend the current job (send a SIGTSTP signal)

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes can be sent to web@petercv.com for new content to be published or updated

Ajay Sangale	Chapter 1
Ajinkya	Chapter 20
Alessandro Mascolo	Chapters 11 and 26
Alexej Magura	Chapters 9, 12, 36 and 61
Amir Rachum	Chapter 8
Anil	Chapter 1
anishsane	Chapter 5
Antoine Bolvy	Chapter 9
Archemar	Chapter 9
Arronical	Chapter 12
Ashari	Chapter 36
Ashkan	Chapters 36 and 43
Batsu	Chapter 17
Benjamin W.	Chapters 1, 9, 12, 15, 24, 31, 36, 46 and 47
binki	Chapter 21
Blachshma	Chapter 1
Bob Bagwill	Chapter 1
Bostjan	Chapter 7
BrunoLM	Chapter 14
Brydon Gibson	Chapter 9
Bubblepop	Chapters 1, 5, 8, 12 and 24
Burkhard	Chapter 1
BurnsBA	Chapter 22
Carpetsmoker	Chapter 47
cb0	Chapter 28
Chandrahas Aroori	Chapter 6
chaos	Chapter 9
charneykaye	Chapter 50
chepner	Chapters 15, 27 and 46
Chris Rasys	Chapter 34
Christopher Bottoms	Chapters 1, 3 and 5
codeforester	Chapter 12
Cody	Chapter 66
Colin Yang	Chapter 1
Cows quack	Chapter 30
CraftedCart	Chapter 1
CrazyMax	Chapter 64
criw	Chapter 36
Daniel Käfer	Chapter 67
Danny	Chapter 1
Dario	Chapters 28, 36 and 55
David Grayson	Chapter 9
Deepak K M	Chapter 20
deepmax	Chapter 25
depperm	Chapters 4 and 35
dhimanta	Chapter 62
dimo414	Chapter 14

dingalapadum	Chapters 7 and 16
divyum	Chapters 1 and 14
DocSalvager	Chapter 10
Doctor J	Chapter 28
DonyorM	Chapter 10
Dr Beco	Chapter 36
Dunatotatos	Chapter 51
Echoes_86	Chapter 17
Edgar Rokjān	Chapter 10
edi9999	Chapter 14
Eric Renouf	Chapter 9
fedorqui	Chapters 12, 15, 17, 20, 28 and 34
fifaltra	Chapters 8 and 53
Flows	Chapter 18
Gavyn	Chapters 9, 26, 33 and 36
George Vasiliou	Chapters 9, 15 and 58
Gilles	Chapters 21 and 22
glenn jackman	Chapters 1, 4, 5 and 7
Grexis	Chapter 15
Grisha Levit	Chapter 36
gzh	Chapter 10
hedgar2017	Chapters 9, 15 and 22
Holt Johnson	Chapter 4
I0_ol	Chapter 64
Iain	Chapters 4 and 20
IamaTacos	Chapter 35
Inanc Gumus	Chapter 1
Inian	Chapters 17 and 28
intboolstring	Chapters 4, 5 and 7
Jahid	Chapters 1, 5, 9, 10, 12, 14, 15, 17, 20, 21, 22, 23, 30, 34, 39, 43, 44 and 45
James Taylor	Chapter 23
Jamie Metzger	Chapter 31
jandob	Chapter 29
janos	Chapters 7, 10, 12, 14, 20 and 24
Jeffrey Lin	Chapter 49
JepZ	Chapter 3
jerblack	Chapter 12
Jesse Chen	Chapters 15, 26 and 45
JHS	Chapters 7, 19 and 67
jimsug	Chapter 24
John Kugelman	Chapter 12
Jon	Chapter 63
Jon Ericson	Chapter 9
Jonny Henly	Chapter 4
jordi	Chapter 48
Judd Rogers	Chapters 9 and 67
Kelum Senanayake	Chapter 23
ksoni	Chapter 30
leftaroundabout	Chapter 17
Leo Ufimtsev	Chapter 33
liborm	Chapter 9
lynxlynxlynx	Chapter 43
m02ph3u5	Chapter 67

markjwill	Chapter 12
Markus V.	Chapter 4
Mateusz Piotrowski	Chapter 12
Matt Clark	Chapters 1, 9, 14, 17, 19 and 23
mattmc	Chapters 36 and 65
Michael Le Barbier Grünewald	Chapter 14
Mike Metzger	Chapter 8
miken32	Chapters 9 and 10
Misa Lazovic	Chapters 4 and 30
Mohima Chaudhuri	Chapters 18 and 41
nautical	Chapter 34
NeilWang	Chapter 12
Neui	Chapter 8
Ocab19	Chapter 58
ormaaj	Chapter 12
Osaka	Chapter 4
P.P.	Chapter 38
Pavel Kazhevets	Chapter 25
Peter Uhnak	Chapter 31
phs	Chapter 47
Pooyan Khosravi	Chapter 9
Rafa Moyano	Chapter 42
Reboot	Chapter 42
Riccardo Petraglia	Chapter 8
Richard Hamilton	Chapters 4, 16, 41 and 57
Riker	Chapters 1 and 40
Roman Piták	Chapter 47
Root	Chapters 5, 8 and 9
Sameer Srivastava	Chapter 8
Samik	Chapters 4, 5, 10, 12, 14 and 37
Samuel	Chapter 5
Saqib Rokadia	Chapter 67
satyanarayan rao	Chapter 1
Scroff	Chapter 66
Sergey	Chapter 14
sjsam	Chapters 1 and 32
Sk606	Chapters 8, 12 and 33
Skynet	Chapter 45
SLePort	Chapters 5 and 10
Stephane Chazelas	Chapters 15 and 36
Stobor	Chapter 20
suleiman	Chapter 59
Sundeep	Chapter 1
Sylvain Bugat	Chapters 2, 4, 9, 14 and 15
Thomas Champion	Chapter 56
Tim Rijavec	Chapter 25
TomOnTime	Chapter 47
Trevor Clarke	Chapter 1
tripleee	Chapters 1, 5, 14, 17 and 36
tversteeg	Chapter 30
uhelp	Chapters 2, 7, 13, 20, 31, 36, 47, 48, 52, 53 and 54
UNagaswamy	Chapters 12, 13 and 60

user1336087	Chapters 1 and 26
vielmetti	Chapter 5
vmaroli	Chapter 39
Warren Harper	Chapter 9
Wenzhong	Chapter 30
Will	Chapters 12, 15 and 21
Will Barnwell	Chapter 24
William Pursell	Chapters 1, 36 and 49
Wojciech Kazior	Chapter 36
Wolfgang	Chapter 9
xhienne	Chapter 5
ymbirtt	Chapter 15
zarak	Chapters 8, 24 and 31
Zaz	Chapter 1
Мона_Сах	Chapter 28
南山竹	Chapters 1, 5, 9, 12 and 17